Journal of Logical and Algebraic Methods in Programming 105 (2019) 1–27

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Time analysis of actor programs ☆

Cosimo Laneve [a], Michael Lienhardt [b], Ka I Pun [c,d], Guillermo Román-Díez [e,*]

[a] University of Bologna/INRIA, Italy
[b] ONERA – The French Aerospace Lab, Palaiseau, France
[c] Western Norway University of Applied Sciences, Norway
[d] University of Oslo, Norway
[e] Universidad Politécnica de Madrid, Spain

ARTICLE INFO

Article history:
Received 23 March 2018
Received in revised form 9 February 2019
Accepted 17 February 2019
Available online 21 February 2019

Keywords:
Time analysis
Behavioral types
Resource analysis

ABSTRACT

This paper proposes a technique for estimating the computational time of programs in an actor model, which is intended to serve as a compiler target of a wide variety of actor-based programming languages. We define a *compositional* translation function returning *cost equations*, which are fed to an automatic off-the-shelf solver for obtaining the time bounds. Our approach is based on a new notion of *synchronization sets*, which captures possible difficult synchronization patterns between actors and helps make the analysis efficient and precise. The approach is proven to correctly over-approximate the worst computational time of an actor model of concurrent programs. Our technique is complemented by a prototype analyzer that returns upper bound of costs for the actor model.

1. Introduction

Time computation for programs running on mainstream architectures, for example, multicore, distributed systems or cloud, is intricate and demanding as the execution of a process may be indirectly delayed by other processes running on different machines due to synchronizations. The computational time of programs is particularly relevant in cloud architectures, where services are bound by so-called *service-level agreements* (SLAs), which regulate the costs in time and assign penalties for their infringements [7]. In particular, the service providers need guarantees that the services meet the SLA, for example in terms of the end-user response time, by deciding on a resource management policy, and by determining the appropriate number of virtual machine instances (or containers) and their parameter settings (e.g., their CPU speeds).

In this paper we propose a technique for estimating the computational time of programs in an actor model. This model is intended to serve as a compiler target of a wide variety of actor-based programming languages, such as object-oriented ones, including `Java` and `C#`, which are used in cloud architectures. Our technique aims at (and in fact, has been developed for) helping service providers to select resource management policies in *a correct way*, before actually deploying the service.

☆ This work has been partially supported by Comunidad de Madrid as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union, by the Spanish MINECO project TIN2015-69175-C4-2-R and the *SIRIUS Centre for Scalable Data Access* (http://www.sirius-labs.no).

* Corresponding author.
E-mail addresses: cosimo.laneve@unibo.it (C. Laneve), michael.lienhardt@onera.fr (M. Lienhardt), Violet.Ka.I.Pun@hvl.no (K. Pun), guillermo.roman@upm.es (G. Román-Díez).

https://doi.org/10.1016/j.jlamp.2019.02.007
2352-2208/

Several techniques have been proposed for analyzing the computational time of sequential programs. See for example [3,6,10,11,17] and Section 7. Our approach is similar to [15], where a statically typed intermediate language has been defined in order to verify safety properties and certify code optimizations. Different from [15], our language, called `alt`, short for *actor language with time*, is concurrent, and contains an operation defining the number of processing cycles required to be computed, called *wait*(n) (similar to the `sleep`(n) operation in `Java`). In order to analyze the computational time of `alt`, we define an algorithm that returns a set of cost equations that are adequate for a solver. We demonstrate that the solution of the cost equations over-approximates the computational time of the `alt` program in input. Given these results, estimating the computational time of a program in some programming languages amounts to defining a compilation into an `alt` program (and demonstrating its correctness).

The presented work builds upon a previous article by the authors [9], where it captures the actor models of actor-based programming languages in terms of *behavioral types*, and obtains the computational time of the corresponding program by feeding the cost equations produced by a translation function to a off-the-shelf solver, namely the `CoFloCo` solver [8]. However, this technique proposed in [9] has a very severe constraint: invocations were admitted only either on the same actor or on newly created ones, i.e., no invocation on parameters. For instance, according to this constraint, an invocation to a method `inner`(y, x), where the first parameter is the actor executing the method, cannot occur in the body of a method `outer`(x, y). The challenge is that, in this case, computing the cost of `outer`(x, y) requires to know whether there is a synchronization between actors x and y. In case there is, one has to consider that `inner`(y, x) might be delayed by other methods running on y, which might be independent from `outer`(x, y).

This paper focuses on overcoming this issue. We first compute *synchronization sets* of actors, which are actors that potentially might interfere with the executions of each other. We then compose the cost of an invocation with the cost of the callee in two ways: (1) it is *added*, corresponding to *sequential compositions*, if the arguments of the invocation and those of the caller are in the same synchronization set; (2) it is the *maximum value*, corresponding to *parallel composition*, otherwise. We then define a new translation function that takes the synchronization sets of an `alt` program into account and returns the corresponding set of cost equations.

The translation of `alt` programs into the solver input code [2,8] has been prototyped and can be experimented (see Section 6). This tool, together with the compiler we have defined in [9], allows us to automatically compute the cost of programs in `ABS`, a prototype language for programming the cloud [13] that is an extension of `alt`. Experimental results show that our technique is very precise when computing the cost of a number of typical distributed patterns, such as *fork-join* or *map-reduce*.

Paper overview. The `alt` language is defined in Section 2 and we discuss in Section 3 the issues in estimating the computational time. Section 4 explains the analysis of computational time by means of a translation function that returns cost equations, and Section 5 discusses the properties of the translation. In Section 6, we illustrate the conversion of the translation output to the adequate form for an off-the-shelf solver, present our prototype, together with the experimental results of our approach and a comparison with other tools. In Section 7, we discuss the related work and deliver concluding remarks in Section 8.

2. The language `alt`

In this section, we define the syntax and the semantics of `alt` and illustrate the language with some examples.

Syntax. We use several disjoint sets of *names* that, for convenience, we address taking different notations or identifiers: *method names*, ranged over by $\mathtt{m}, \mathtt{m}', \cdots$; *actor* names ranged over by $x, y, \cdots$; *natural* names $u, w, a, b, \cdots$; *future names* ranged over by $f, g, h, \cdots$. The notations $\overline{x}$ and $\overline{u}$ denote a possibly empty sequence of actor and natural names, respectively; $\overline{p}$ ranges over sequences whose elements may be either actor or natural names. An `alt` program

$$\left(\mathtt{m}_1(\overline{p}_1) = s_1, \cdots, \mathtt{m}_n(\overline{p}_n) = s_n, \mathtt{main}(x, \overline{u}) = s\right)$$

is a sequence of method definitions of the form $\mathtt{m}_i(\overline{p}_i) = s_i$. The last method is the main method, where x is the actor name executing the main body and $\overline{u}$ are natural names. Statements s and expressions e are defined by the following syntax:

$$\begin{array}{rcll} s & ::= & 0 \mid \nu x; s \mid \nu f\colon \mathtt{m}(\overline{e}); s \mid f^{\checkmark}; s \mid \mathit{wait}(e_n); s & \\ e & ::= & e_n \mid x & \\ e_n & ::= & k \mid u \mid e_n + e_n & (k \text{ are natural constants}) \end{array}$$

The syntax of `alt` statements is quite basic: 0 indicates a terminated statement; νx is the creation of a new actor x; $\nu f\colon \mathtt{m}(\overline{e}); s$ creates a new task of the method $\mathtt{m}$ that is associated to the future f; the task *starts in parallel* with the continuation s. The term $f^{\checkmark}$ performs the synchronization of the task associated to f. This may cause the (busy) waiting for the termination of the task. The statement *wait*(e_n), where e_n is a natural expression, represents the advance of e time units. This is the only term in our model that consumes time (a.k.a. that generates a cost). Note that term e_n in *wait*(e_n) contains constants and natural names, including the arithmetic binary operator $+$ between them. These expressions are also known as *Presburger arithmetics*, which is a decidable fragment of Peano arithmetics containing only addition.

The restriction is necessary because the analysis in Section 4 cannot deal with generic expressions. Note that, in general, expressions e also include actor names (in the syntax of e, x also ranges over actor names), as in the method invocations.

Method definitions have the form $\mathtt{m}(\overline{p}) = \nu z_1\,;\cdots;\nu z_k\,; s$ such that all actors can only be created in the beginning of the method. Additionally, the method definition $\mathtt{m}(\overline{p}) = s$ binds names $\overline{p}$ to the body s. We assume that $\overline{p}$ has always a fixed pattern: $\overline{p} = x, \overline{y}, \overline{u}$, namely $\overline{p}$ is a nonempty sequence where the first element is *always* an actor name indicating the callee of the method, $\overline{y}$ are actor names and $\overline{u}$ are natural names. We assume that actor names $x, \overline{y}$ do not clash with names $z_1, \cdots, z_k$. Therefore, method invocations $\nu f\colon \mathtt{m}(\overline{e})$ have the pattern $x', \overline{y'}, \overline{e'}$. We further assume that `alt` has a simple type system that verifies the type correctness of method applications. Statements $\nu x\,; s$ and $\nu f\colon \mathtt{m}(\overline{e})\,; s$ link the names x and f in the continuations s. For the notion of bound name and the one of free name, we use the standard operations of alpha-conversion and substitution. It is also assumed that method names $\mathtt{m}_1, \cdots, \mathtt{m}_n$ in program declarations are pairwise different.

Example 2.1. To illustrate `alt`, consider the program:

```
1  main(x, a, b) =
2    νy; νz;
3    νf: timer(y, a);
4    f✓;
5    νg: timer(z, b);
6    g✓;

8  main′(x, a, b) =
9    νy; νz;
10   νf: timer(y, a);
11   νg: timer(z, b);
12   f✓;
13   g✓;

15 timer(x, n) =
16   wait(n);
```

Consider method `main` which is executed by actor x. Note that this actor is automatically created when the main method starts to run. After creating two actors y and z at line 2, `main` invokes $\mathtt{timer}(y, a)$ and continues its execution. By binding this invocation to the future name f and synchronizing it immediately afterwards at line 4, it is possible to delay the execution of the caller actor x until the termination of $\mathtt{timer}(y, a)$. In this case, since y has been just created, it will immediately execute $\mathtt{timer}(y, a)$ and return (this is not true in general because y might have other tasks running). Therefore, x will wait *exactly* a time units (effect of the instruction *wait*(n) at line 16) before continuing. The continuation spawns a task $\nu g\colon \mathtt{timer}(z, b)$ at line 5 on actor z and waits for its termination, which, for the same reasons as before, will occur after b time units. Overall, the computational time of `main` is $a + b$. Observe that the time is the same when line 5 is replaced by $\nu g\colon \mathtt{timer}(y, b)$, e.g., spawning the thread on y instead of z.

In the case of `main′`, the two invocations to `timer` are performed one after the other, without any synchronization in between. Thus the two tasks are executed in parallel, and as they are both synchronized at lines 12 and 13, the computational time of `main′` is $max(a, b)$. Observe that if line 11 is replaced by $\nu g\colon \mathtt{timer}(y, b)$, e.g., spawning the thread on y instead of z, the computational time becomes $a + b$. □

Two features of `alt` ease our theoretical development: (*i*) `alt` actors are *stateless* and (*ii*) methods do not return values. In fact, computing the cost of stateful actor programs requires a leap of the theory developed in this contribution because it is necessary to trace the state of the fields. Similarly, when methods return a value, it is necessary to estimate the value (is it an old actor or a new one? if it is a natural number, how large it is?) in order to have sensible cost computations.

Semantics. The semantics of `alt` is a transition system whose states are *configurations* cn that are defined as follows.

$$cn \;::=\; act(x,p,q) \;\mid\; fut(f,val) \;\mid\; invoc(x,f,\mathtt{m},\overline{v}) \;\mid\; cn\;cn$$

$$val \;::=\; \bot \mid \top \qquad\qquad p \;::=\; s\,; f \mid f \mid \mathtt{0}$$
$$v \;::=\; x \mid f \mid k \qquad\qquad q \;::=\; \emptyset \mid s\,; f \mid q\;q$$

The notation $s\,; f$ represents the statement s where the tailing `0` is replaced by the future name f. We use p to range over $s\,; f$, f, and `0`. A *configuration* cn is a nonempty set of actors, invocation messages and futures. The associative and commutative union operator on configurations is denoted by whitespace. An actor is written as $act(x, p, q)$, where x is the identity of the actor, p is the *active task*, and q is a pool of either *suspended* or *waiting tasks*. An *invocation message* is denoted as $invoc(x, f, \mathtt{m}, \overline{v})$, where x is the callee, f the future to which the call is bound, $\mathtt{m}$ the method name, and $\overline{v}$ the set of parameter values of the call. Configurations also include *futures*, denoted as $fut(f, val)$, where f is the future identity and val indicates whether f has already been computed, written as $\top$, otherwise $\bot$.

The *transition rules* of `alt` are given in Fig. 1. We use an auxiliary function to bind invocations to the corresponding method bodies. Let $\mathtt{m}(x, \overline{z}) = s$ be an `alt` method. Then

$$\mathrm{bind}(y, \mathtt{m}, \overline{v}) = s\{^{y,\,\overline{v}}/_{x,\,\overline{z}}\}\,.$$

$$\frac{cn \to cn'}{cn\ cn'' \to cn'\ cn''}\ (\textsc{Context}) \qquad \frac{z' = \text{fresh}()}{act(x, \nu z\,; p, q) \to act(x, p\{^{z'}/_{z}\}, q)\ \ act(z', 0, \emptyset)}\ (\textsc{New})$$

$$act(x, f^{\checkmark}\,; p, q)\ \ fut(f, \top) \to act(x, p, q)\ \ fut(f, \top)\ (\textsc{Get-True}) \qquad act(x, f^{\checkmark}\,; p, q)\ \ fut(f, \bot) \to act(x, 0, q \cup (f^{\checkmark}\,; p))\ \ fut(f, \bot)\ (\textsc{Get-False})$$

$$\frac{[\![\overline{e}]\!] = \overline{v} \quad f' = \text{fresh}()}{act(x, \nu f\colon \mathtt{m}(z, \overline{e})\,; p, q) \ \to\ act(x, p\{^{f'}/_{f}\}, q)\ \ invoc(z, f', \mathtt{m}, \overline{v})\ \ fut(f', \bot)}\ (\textsc{Async-Call})$$

$$\frac{s = \text{bind}(x, \mathtt{m}, \overline{v})}{act(x, p, q)\ \ invoc(x, f, \mathtt{m}, \overline{v}) \to act(x, p, q \cup s\,; f)}\ (\textsc{Bind-Fun}) \qquad \frac{[\![e]\!] = 0}{act(x, wait(e)\,; s, q) \to act(x, s, q)}\ (\textsc{Wait-0})$$

$$act(x, 0, q \cup p) \to act(x, p, q)\ (\textsc{Activate}) \qquad act(x, f, q)\ \ fut(f, \bot) \to act(x, 0, q)\ \ fut(f, \top)\ (\textsc{Return}) \qquad \frac{strongstable_t(cn)}{cn \xrightarrow{t} \Phi(cn, t)}\ (\textsc{Tick})$$

Fig. 1. Transition rules.

We also use the function fresh() to return either a fresh actor name or a fresh future name. Finally, we use an *evaluation function* $[\![\cdot]\!]$ of expressions e such that $[\![x]\!] = x$, and $[\![e]\!] = v$ whenever e is a constant natural expression and v is its value; $[\![e]\!]$ is undefined otherwise.

The semantics of `alt` is almost standard. See, for example [14], for a similar semantics for a dialect of pi-calculus with time. We discuss the most relevant rules in the following: actor creation, method invocation, method return and the *wait*(e) statement.

Actor creation is handled by rule New, which extends the configuration with $act(z', 0, \emptyset)$ where z' is a fresh actor identifier. In `alt`, method invocations are *asynchronous*: rule Async-Call creates a new invocation predicate $invoc(z, f', \mathtt{m}, \overline{v})$ and a new unresolved future predicate $fut(f', \bot)$, which is associated to the invocation with a fresh future identifier f'. The invocation predicate will then be bound to the corresponding callee actor as one of the tasks in the task pool (cf. rule Bind-Fun). When a method terminates, rule Return sets the value of the corresponding future f to $\top$. Note that f is added to the end of the method body in rule Bind-Fun. In our model, *wait*(e) is the unique operation that consumes time; that is, time does not advance as long as a *wait*(e) statement is prefixed by some active task. For the trivial case where $e = 0$ (see rule Wait-0), the statement is simply discarded. On the contrary, when a configuration cn reaches a *stable* state, that is, no other transition is possible apart from those evaluating the *wait*(e) statements, time advances until an actor with a non-*wait*(e) statement can proceed. To formalize this semantics, we first introduce the notion of *stability* as follows:

Definition 2.1. Let $t > 0$. A configuration cn is *t-stable*, written as $stable_t(cn)$, if every actor in cn matches one of the following forms:

1. $act(x, wait(e)\,; s\,; f', q)$ with $[\![e]\!] \geq t$,
2. $act(x, 0, q)$ and
   i. either $q = \emptyset$,
   ii. or, for every $s \in q$, $s = f^{\checkmark}\,; p$ and $fut(f, \bot) \in cn$.

A configuration cn is *strongly t-stable*, written as $strongstable_t(cn)$, if it is t-stable and there exists an actor $act(x, wait(e)\,; s, q)$ with $[\![e]\!] = t$.

Note that t-stable (and consequently, strongly t-stable) configurations cannot progress anymore because every actor is stuck either on a *wait*$(\cdot)$-statement or on an unresolved future.

We then define a function to update a configuration cn with respect to a time value t.

$$\Phi(cn, t) = \begin{cases} act(x, wait(k)\,;s, q)\ \Phi(cn', t) & \text{if } cn = act(x, wait(e)\,;s, q)\ cn' \\ & \text{and } k = [\![e]\!] - t \\ act(x, 0, q)\ \Phi(cn', t) & \text{if } cn = act(x, 0, q)\ cn' \\ \\ cn & \text{otherwise.} \end{cases}$$

Together with the rule Tick in Fig. 1, we define the *time progress* of a configuration.

The initial configuration of a program whose main method is $\texttt{main}(x, \overline{u}) = s$ and with invocation $\texttt{main}(z, \overline{k})$ is

$$act(z, s\{\overline{k}/\overline{u}\}\,; f_{start}, \emptyset)$$

where f_{start} is the future identifier for the main body. As usual, $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$ and $\overset{t}{\Rightarrow}$ is $\rightarrow^* \overset{t}{\rightarrow} \rightarrow^*$. A *computation* is $cn \overset{t_1}{\Rightarrow} \cdots \overset{t_n}{\Rightarrow} cn'$, that is, cn' is a configuration reachable from cn with either transitions $\rightarrow$ or $\overset{t}{\Rightarrow}$. When the time labels of transitions are not relevant we also write $cn \Rightarrow^* cn'$.

Definition 2.2. The *computational time of* $cn \overset{t_1}{\Rightarrow} \cdots \overset{t_n}{\Rightarrow} cn'$ is $t_1 + \cdots + t_n$.

The *computational time of a configuration* cn, written $time(cn)$, is the maximum computational time of computations starting at cn. The *computational time* of an `alt` program is the computational time of its initial configuration.

Example 2.2. Consider the following method $\texttt{main}_1$:

```
1  main1(x, a, b) =
2      νy;
3      wait(k0);
4      νf: m1(y);
5      wait(a);
6      νg: m2(y);
7      wait(b);
8      g✓;
9      f✓;

10 m1(x) =
11     wait(k1);
12
13 m2(x) =
14     wait(k2);
```

This method creates a new actor y at line 2 and spawns two tasks on it at lines 4 and 6, respectively, where the two tasks will execute in parallel with $\texttt{main}_1$. Their terminations are synchronized at lines 8 and 9 by means of $g^{\checkmark}$ and $f^{\checkmark}$. Note that $\texttt{main}_1$ takes its natural number arguments as the input parameter of the $wait(\cdot)$-operations at lines 5 and 7, while the one at line 3 uses a constant k_0. Thus, the computational time of $\texttt{main}_1$ depends on the concrete values of a and b. The other $wait(\cdot)$-statements in the example are executed with some constants k_0, k_1 and k_2. □

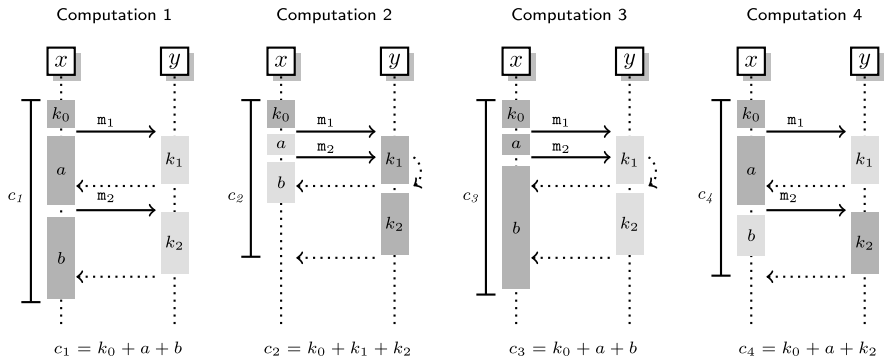
Our semantics does not exclude behaviors of methods that perform infinite actions without consuming time (preventing rule Tick to apply), such as $\texttt{foo}(x) = \nu f\colon \texttt{foo}(x)\,; f^{\checkmark}$. This kind of behaviors are well-known in the literature (*cf.* Zeno behaviors, see [14]) and they may be easily excluded from our analysis by constraining recursive invocations to be prefixed by a $wait(e)$-statement, where e is a not-zero natural. A similar paradoxical behavior is $\texttt{gee}(x) = \nu y\,; \nu f\colon \texttt{gee}(y)\,; f^{\checkmark}$ that spawns the new task on a new actor, thereby creating an unbounded number of actors.

We finally remark that the composite effect of Activate and Get-False might produce infinite computations that do not make progress, and no time is passing (an actor can repeatedly activate a blocked task and then suspend it again). However, assuming fairness in computations (an enabled task will eventually be executed), it is possible to demonstrate that these behaviors are not possible in `alt`.

3. The challenges of cost computation for `alt` programs

Computing the time of `alt` programs is challenging. In this section, we illustrate the difficulties by discussing four examples. These difficulties range from determining the tasks that *cannot execute in parallel* – therefore the cost of each of these tasks must be *added* together –, to understanding the tasks that *are executed in parallel* – thus, the *maximum value* of the cost of the tasks is selected – or to analyzing the whole spectrum of scheduling policies because they might also affect the computational time.

Example 3.1. The method $\texttt{main}_1$ in Example 2.2 invokes $\texttt{m}_1$ and $\texttt{m}_2$ on an actor that is different from the caller. The following graphical representations illustrate four different computations that are obtained by choosing different values of a and b. Rectangles in dark gray represent the largest segment of time that are used for computing the maximal cost of the computations, which are denoted by the names c_1, c_2, c_3 and c_4, respectively.



Computation 1 represents an execution where $a > k_1$, which leads to the execution of $\mathtt{m}_2$ on actor y starts *after* the termination of $\mathtt{m}_1$. In this case, the executions of *wait*(b) and $\mathtt{m}_2$ run in parallel. Thus, the cost of this computation is $k_0 + a + max(b, k_2)$ (the expression c_1 in the figure reflects the case where $b \geq k_2$). Computation 2 and Computation 3 describe computations where $a < k_1$. This leads to delaying $\mathtt{m}_2$ until the execution of $\mathtt{m}_1$ finishes. In Computation 2, since $a + b < k_1 + k_2$, actor x has to wait for the termination of $\mathtt{m}_2$. On the contrary, in Computation 3, $\mathtt{m}_2$ returns before actor x finishes the execution of *wait*(b). Finally, Computation 4 illustrates an execution where both x and y have to wait either for a method to terminate and return or for executing a new task.

As our analysis aims at finding a sound approximation of time needed for any computation of the program, the cost of $\mathtt{main}_1$ is over-approximated by the expression

$$c_{\mathtt{main}_1} = k_0 + max(a, k_1) + max(b, k_2)$$

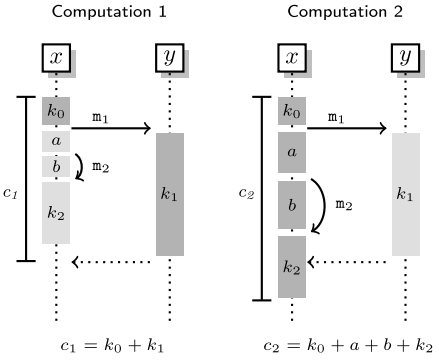
For Computation 1, Computation 2 and Computation 4, $c_{\mathtt{main}_1}$ captures precisely the execution time. Note that the sub-expression $k_0 + max(a, k_1)$ determines the starting time of $\mathtt{m}_2$ on actor y. However, $k_0 + max(a, k_1)$ is an over-approximation of the starting time of the statement *wait*(b) at line 7 of $\mathtt{main}_1$, ending up in a non-precise result for Computation 3 as $c_{\mathtt{main}_1} > c_3$. □

Another challenging issue to cope with is when a task is delayed due to an unresolved future (see rule GET-FALSE). In this situation, the carrier actor may start the execution of the pending tasks and may reschedule the initial task after the termination of the other ones.

Example 3.2. Consider following method $\mathtt{main}_2$, which is similar to $\mathtt{main}_1$ in Example 2.2, except that $\mathtt{m}_2$ is invoked on the callee actor x. The computations on the right show two possible executions of $\mathtt{main}_2$, depending on the values of a and b.

```
1  main2(x, a, b) =
2      νy;
3      wait(k0);
4      νf: m1(y);
5      wait(a);
6      νg: m2(x);
7      wait(b);
8      g✓;
9      f✓;

10 m1(x) =
11     wait(k1);
12
13 m2(x) =
14     wait(k2);
```



In Computation 1, the synchronization $g^{\checkmark}$ at line 8 happens before $\mathtt{m}_1$ terminates. In this case, the cost of executing $\mathtt{m}_2$ does not contribute to the overall cost c_1. On the contrary, in Computation 2, the cost of executing $\mathtt{m}_1$ does not contribute to the overall cost c_2. The reason is that the values of a and b in this computation are larger than Computation 1, which consequently results in longer time for executing *wait*(a), *wait*(b) and $\mathtt{m}_2$ (i.e., *wait*(k_2)) on actor x than $\mathtt{m}_1$ on actor y. The expression

$$c_{\mathtt{main}_2} = k_0 + max(a + b + k_2, k_1)$$

captures the worst case computational time of $\mathtt{main}_2$, which over-approximates c_1 and c_2. □

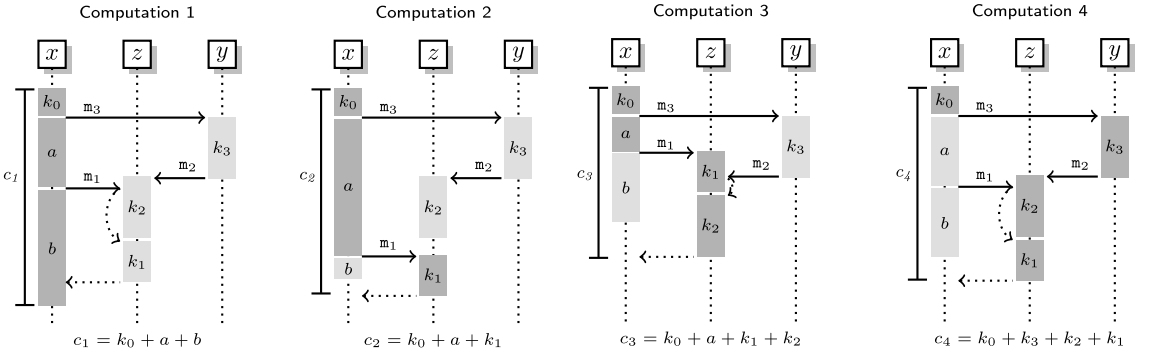
A more complex scenario can be illustrated by having two actors x and y spawning tasks on a third actor z. In this case, the execution of tasks on x and y may delay the starting time of those spawned on z. As a consequence, it may lead to the postponement of the synchronizations on x and y. Determining the delays of synchronizations is crucial, as discussed in the following example.

Example 3.3. Consider the following method $\mathtt{main}_3$ that creates two actors y and z at line 2, then invokes $\mathtt{m}_3$ on y at line 4 and $\mathtt{m}_1$ on z at line 6. The actor z is given as a parameter in the invocation of $\mathtt{m}_3$ (line 4) and becomes the carrier of $\mathtt{m}_2$ (line 18).

```
1  main3(x, a, b) =
2      νy;  νz;
3      wait(k0);
4      νf: m3(y, z);
5      wait(a);
6      νg: m1(z);
7      wait(b);
8      g✓;
9      f✓;

10 m1(x) =
11     wait(k1);
12
13 m2(x) =
14     wait(k2);
15
16 m3(x, y) =
17     wait(k3);
18     νh: m2(y);
19     h✓;
```

The figure below shows four different computations of $\mathtt{main}_3$ that correspond to various values of a, b, k_1, k_2 and k_3.



In Computation 1, the cost is the sum of the costs of the *wait*$(\cdot)$ statements in $\mathtt{main}_3$. This cost does not include the cost of any task executed on y and z as they are running in parallel with $\mathtt{main}_3$ without any delay.

Computation 2 combines the costs from actors x and z but not from y because $a > k_2 + k_3$. Observe that the time needed to start $\mathtt{m}_1$ is determined by the expression $k_0 + a$; observe also that *wait*(k_3) and *wait*(k_2) execute sequentially although they run on different actors (y and z). Note that, *wait*(b) and *wait*(k_1) are executed in parallel on x and z, and since $k_1 > b$, the maximum cost is expressed as $c_2 = k_0 + a + k_1$.

In Computation 3, we assume $a < k_3$ such that the execution of $\mathtt{m}_1$ on z starts before the invocation of $\mathtt{m}_2$ on z. Thus, the execution of $\mathtt{m}_2$ is delayed until $\mathtt{m}_1$ has finished. The cost of Computation 3 is calculated by the expression $c_3 = k_0 + a + k_1 + k_2$.

Computation 4 highlights a possible dependency among tasks executed on actors y and z. The cost expression for this case is $c_4 = k_0 + k_3 + k_2 + k_1$, describing a sum to which all three actors contribute (k_0 on x, k_3 on y and $k_2 + k_1$ on z). The reason is that $a > k_3$ where the statement *wait*(k_3) on y (line 17) directly determines the starting time of $\mathtt{m}_2$ on z, and which in this case delays the execution of $\mathtt{m}_1$ on z as a consequence.

One remarkable difference between Computation 2 and Computation 4 is the time taken before starting $\mathtt{m}_1$ on actor z. In Computation 2, the cost $k_0 + k_3$ is smaller than $k_0 + a$, causing that $\mathtt{m}_2$ is executed before $\mathtt{m}_1$ on z. When there are two actors whose tasks could be dependent on each other, their costs must be considered as if the tasks are executed sequentially, that is, their costs are summed up. Thus, the relation between the tasks executed on y and on z must be considered in order to over-approximate all possible computations of $\mathtt{main}_3$. Considering all possible values of a and b, the cost expression that over-approximates the cost of executing $\mathtt{main}_3$ is

$$c_{\mathtt{main}_3} = k_0 + max(a, k_2 + k_3) + max(b, k_1)$$

Observe that $c_{\mathtt{main}_3}$ precisely approximates the overall cost of Computation 2 and Computation 4. However, to safely approximate the latest starting time of $\mathtt{m}_1$ in case $a > k_3$, the subexpression $max(a, k_2 + k_3)$ returns $k_2 + k_3$. Thus, we have that

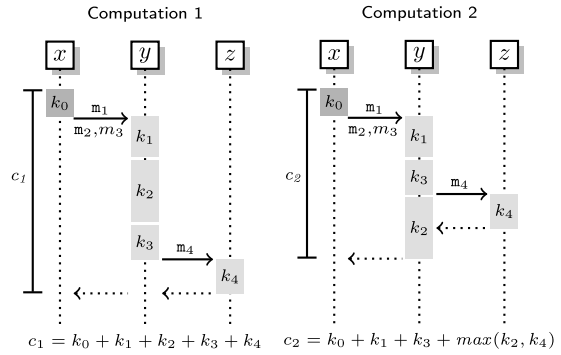
$c_{\mathtt{main}_3} > c_3$, indicating that this expression is over-approximating the cost of Computation 3. Similarly, Computation 1 is also over-approximated by c_3 as it captures that the starting time of $\mathtt{m}_1$ is $k_0 + k_2 + k_3$, when its starting time is $k_0 + a$. □

This last example suggests that, when a method invocation has several actors as arguments, the pools of pending tasks of these actors can have implicit and possibly complex dependencies, i.e., each of them may in principle affect the task pools of the others by delaying the execution of the tasks therein. We say that these actors belong to the same *synchronization set*. It is crucial to note that this synchronization relation is *transitive*. For instance, if x and y belong to the same synchronization set at a program point and if a new synchronization relation between y and z is established by a subsequent invocation, then actors x, y and z belong to the same synchronization set as well.

The final example illustrates how the scheduling might also affect the computational time of the `alt` programs.

Example 3.4. The following example shows the method $\mathtt{main}_4$ that spawns three tasks in actor y in parallel – $\mathtt{m}_1$, $\mathtt{m}_2$ and $\mathtt{m}_3$ – that will be in the pool of pending tasks of y. In turn, $\mathtt{m}_3$ spawns $\mathtt{m}_4$ in a newly created actor z.

```
1  main4(x) =
2      νy;
3      wait(k0);
4      νf: m1(y);
5      νg: m2(y);
6      νh: m3(y);
7      h✓; g✓; f✓;
8
9  m1(x) =
10     wait(k1);
11 m2(x) =
12     wait(k2);
13
14 m3(x) =
15     νz;
16     wait(k3);
17     νf: m4(z);
18     f✓;
19
20 m4(x) =
21     wait(k4);
```



We have also highlighted two possible computations of $\mathtt{main}_4$. Computation 1 illustrates an execution where task $\mathtt{m}_2$ is scheduled before $\mathtt{m}_3$. Therefore, the execution of $\mathtt{m}_4$, spawned by $\mathtt{m}_3$, is executed when $\mathtt{m}_2$ is finished. This scheduling leads to the computational time c_1 that adds the cost of all methods. Additionally, note that different scheduling orders in y affect the execution of $\mathtt{m}_4$ in z. On the contrary, in Computation 2 the execution of $\mathtt{m}_3$ is scheduled before $\mathtt{m}_2$. In this case, after executing $wait(k_3)$ at line 16, $\mathtt{m}_4$ is spawned and its return is synchronized at line 18. Note that, as $\mathtt{m}_2$ is in the pool of pending tasks of y, it will be executed in parallel with $\mathtt{m}_4$. The computational time of Computation 2 will be smaller: $c_2 = k_0 + k_1 + k_3 + max(k_2, k_4)$. For this program, the expression that over-approximates the cost of executing $\mathtt{main}_4$ needs to sum up the cost of all methods:

$$c_{\mathtt{main}_4} = k_0 + k_1 + k_2 + k_3 + k_4$$

Since our technique is sound (see Section 5 for details), i.e., it returns an over-approximation for every possible computational time, it takes into account every scheduling order when multiple tasks can be pending in the same pool. □

4. The analysis of `alt` programs

This section defines the translation of an `alt` program into a *cost program*, i.e., a set of cost equations $m(\overline{u}) = exp$, where m is a (cost) function symbol and exp is an expression that may contain (cost) function applications of the form:

$$exp ::= k \mid u \mid exp + exp \mid m(\overline{e}) \mid max(exp, exp)$$

In particular, cost expressions are additions of the following type of elements: natural numbers k, natural names u, $m(\overline{e})$ which refers to the cost of executing method $\mathtt{m}$, but only considering the natural expressions in $\overline{e}$; or $max(exp, exp')$, which returns the maximum between two cost expressions exp and exp'.

Given an `alt` program $\mathcal{P}$, the analysis iterates independently over each method definition $\mathtt{m}(x, \overline{y}, \overline{u}) = s$ in $\mathcal{P}$ and translates it into a cost equation of the form $m(\overline{u}) = exp$, where m corresponds to the method name $\mathtt{m}$ and exp encodes a cost function which corresponds to an upper bound of the time executing $\mathtt{m}$, with respect to the numerical parameters $\overline{u}$. As hinted in our examples, the analysis performs this translation by studying the task pool of every actor involved in the method's execution, identifying an upper bound for the ending time of each of its tasks, which in turn give raise to an upper bound to the computational time of the method itself.

In order to manage the complexity of studying task pools and the different scheduling possibilities as described in Section 3, our analysis uses two major structures, *synchronization schema* and *accumulated costs*. The *synchronization schema* of a method gives the synchronization set of every actor involved in that method's body. As described in Example 3.3, actors in the same synchronization set interact outside the scope of the analyzed method, which means that the ending time of

```
1  main(x, a, b) =
2     νy;  νz;  νw;
3     wait(k0);
4     νf: m1(x);
5     νg: m3(y, z);
6     wait(a);
7     νh: m4(w);
8     νi: m2(z);
9     i✓;
10    wait(b);
11    h✓;
12    f✓; g✓;
13 m1(z) =
14    wait(k1);
15
16 m2(z) =
17    wait(k2);
18
19 m3(y, z) =
20    wait(k3);
21    νf: m4(z);
22    f✓;
23
24 m4(z) =
25    wait(k4);
```

Fig. 2. Running example of a `alt` program.

a task in one actor's task pool depends somehow on the task pools and its scheduling orders of other actors in the same synchronization set. The analysis over-approximates this implicit dependency by considering the pools of all these actors as if they were one unique pool. An *accumulated costs* is an abstraction of a task pool used to get a good upper bound for the ending time of a computation. The main difficulty in identifying such an upper bound is to determine the starting time of a computation, which depends on the state of the task pool itself and on the time at which the corresponding method call was performed.

4.1. Synchronization schema

A *synchronization set*, ranged over by A, B, $\cdots$, is a set of actor names whose tasks have implicit dependencies, that is, the tasks of these actors may reciprocally affect the task pools of the other actors in the same set by means of method invocations and synchronizations. A *synchronization schema*, ranged over by $\mathcal{S}$, $\mathcal{S}'$, $\cdots$, is a collection of pairwise disjoint synchronization sets. The synchronization schema of a method $\mathtt{m}$ is thus a partition of the actors used in that method into synchronization sets and is constructed as follows.

Definition 4.1 *(Synchronization schema function).* Let $\mathcal{S}$ be a synchronization schema and s be a statement. Let

$$\mathtt{sschem}(\mathcal{S}, s) = \begin{cases} \mathcal{S} \oplus \{y', \overline{z}\} & \text{if } s = \nu f\colon \mathtt{m}(y', \overline{z}, \overline{e}) \\ \mathtt{sschem}(\mathtt{sschem}(\mathcal{S}, s'), s'') & \text{if } s = s'; s'' \\ \mathcal{S} & \text{otherwise} \end{cases}$$

where

$$\mathcal{S} \oplus A = \begin{cases} A & \text{if } \mathcal{S} = \emptyset \\ (\mathcal{S}' \oplus A) \cup A' & \text{if } \mathcal{S} = \mathcal{S}' \cup A' \text{ and } A \cap A' = \emptyset \\ \mathcal{S}' \oplus (A' \cup A) & \text{if } \mathcal{S} = \mathcal{S}' \cup A' \text{ and } A \cap A' \neq \emptyset \end{cases}$$

Let be an `alt` method definition. The *synchronization schema* $\mathcal{S}_{\mathtt{m}}$, where $\mathtt{m}(x, \overline{y}, \overline{u}) = \nu z_1; \cdots; \nu z_k; s$, is the set $\mathcal{S}_{\mathtt{m}} = \mathtt{sschem}(\{\{x, \overline{y}\}\}, s)$. The term $\mathcal{S}(x)$ denotes the synchronization set, which contains x, in the synchronization schema $\mathcal{S}$.

The function $\mathcal{S} \oplus A$ merges a schema $\mathcal{S}$ with a synchronization set A. If none of the actors in A belongs to a set in $\mathcal{S}$, the operation reduces to a simple set union. For example, let $\mathcal{S}$=$\{\{x, y\}, \{z, w\}\}$. Then $\mathcal{S} \oplus \{x, v\}$ is equal to $(\{\{x, y\}\} \oplus \{x, v\}) \cup \{\{z, w\}\}$, which equals $\{\{x, y, v\}, \{z, w\}\}$. It is worth to observe that the same result follows by swapping $\{x, y\}$ and $\{z, w\}$ (because elements of $\mathcal{S}$ are disjoint). In fact $\{\{z, w\}\} \oplus (\{x, y\} \cup \{x, v\}) = \{\{x, y, v\}, \{z, w\}\}$.

The synchronization schema $\mathtt{sschem}(\mathcal{S}, \nu f\colon \mathtt{m}(y, z)) = \{\{x, y, z, w\}\}$ indicates that all the actors are in the same synchronization set. On the contrary, $\mathtt{sschem}(\mathcal{S}, \nu f\colon \mathtt{m}(x, v)) = \{\{x, y, v\}, \{z, w\}\}$, because actors x, v do not belong to $\{z, w\}$.

We also compute $\mathcal{S}_{\mathtt{main}}$, where $\mathtt{main}(x, a, b) = \nu y; \nu z; \nu w; s_{\mathtt{main}}$ is the main method in Fig. 2. This schema is equal to $\{\{x\}, \{y, z\}, \{w\}\}$, that is the task pools of actors y and z may affect each other due to the invocation at line 8, whereas the task pools of x and w are independent from the rest. Observe that, as the body of $\mathtt{m}_3$ does not contain actor creation statements, its synchronization schema is a singleton set that containing only its parameters, e.g., $\mathcal{S}_{\mathtt{m}_3} = \{\{y, z\}\}$.

4.2. Accumulated costs

In order to define (an over-approximation of) the time advancements of tasks in a same pool – e.g. in a same synchronization set – we use an extension of the syntax of *exp*, called *accumulated cost* and noted ε, which is defined as follows:

$$\varepsilon \ ::= exp \ \mid \ \varepsilon \cdot \langle m(\overline{e}), exp\rangle \ \mid \ \varepsilon \parallel exp \ .$$

The term *exp* defines the starting time of a task invoked in an actor that does not belong to the same synchronization set of the carrier. Let this carrier be x in the discussion below and consider an actor y that resides in a different synchronization set than that of x. The term $\varepsilon \cdot \langle m(\overline{e}), exp\rangle$ represents the starting time of a method invoked on actor y. For example, when x invokes a method $\mathtt{m}$ on y using $\nu f\colon \mathtt{m}(y, \overline{z}, \overline{e})$, the accumulated cost of the synchronization set of y is $\varepsilon \cdot \langle m(\overline{e}), 0\rangle$, where ε is the cost up to that point. Time advancements in the task of the carrier x (e.g., with $wait(e')$), which affect the starting time of subsequent method invocations on y, is expressed by adding e' to *exp*; namely, $\varepsilon \cdot \langle m'(\overline{e}), exp\rangle$ becomes $\varepsilon \cdot \langle m'(\overline{e}), exp + e'\rangle$, which means that the starting time of the next method invocation on the synchronization set of y is after the time expressed by ε *plus* the *maximum* between $m'(\overline{e})$ and $exp + e'$. The term $\varepsilon \parallel exp$ expresses the time advancement in the carrier x when a method running on an actor y in a different synchronization set is synchronized. In this case the time advances by the maximum between the current time *exp* of x and the time of y, represented by ε.

Following the semantics described above, we can define the function $[\![\varepsilon]\!]$ that computes the time corresponding to ε, i.e., the (over-approximation of) starting time of the next task in the synchronization set whose cost is ε:

$$[\![exp]\!] = exp \qquad [\![\varepsilon \cdot \langle m(\overline{e}), exp\rangle]\!] = [\![\varepsilon]\!] + max(m(\overline{e}), exp)$$
$$[\![\varepsilon \parallel exp]\!] = max([\![\varepsilon]\!], exp)$$

Example 4.1. The following expressions ε_i indicate the accumulated costs in actor y for some interesting program points of method $\mathtt{main}_1$ in Example 2.2:

3:	$wait(k_0);$	$\varepsilon_3 = 0$
4:	$\nu f\colon \mathtt{m}_1(y);$	$\varepsilon_4 = k_0 \cdot \langle m_1(), 0\rangle$
5:	$wait(a);$	$\varepsilon_5 = k_0 \cdot \langle m_1(), a\rangle$
6:	$\nu g\colon \mathtt{m}_2(y);$	$\varepsilon_6 = k_0 \cdot \langle m_1(), a\rangle \cdot \langle m_2(), 0\rangle$
7:	$wait(b);$	$\varepsilon_7 = k_0 \cdot \langle m_1(), a\rangle \cdot \langle m_2(), b\rangle$

Note that each task creation in y includes a new tuple in the accumulated cost: $\langle m_1(), 0\rangle$ in ε_4 and $\langle m_2(), 0\rangle$ in ε_6. Note also that time advancements local in the carrier, i.e., actor x in Example 2.2, update the last tuple in the accumulated cost with the corresponding expression: a in ε_5 and b in ε_7.

Next, consider the expression $\varepsilon_7 = k_0 \cdot \langle m_1(), a\rangle \cdot \langle m_2(), b\rangle$. Because of the synchronization $g^{\checkmark}$ at line 8 of $\mathtt{main}_1$ we obtain the following cost expression:

$$[\![\varepsilon_7]\!] = k_0 + max(m_1(), a) + max(m_2, b)$$

that over-approximates the worst possible time when g is synchronized. We observe that k_0 is the computational time when $\mathtt{m}_1$ is invoked, $max(m_1(), a)$ expresses that $\mathtt{m}_1$ and $wait(a)$ can be executed in parallel (analogously for $max(m_2(), b)$) and $k_0 + max(m_1(), a)$ corresponds to the worst possible starting time of $\mathtt{m}_2$. It is worth to notice that $[\![\varepsilon_7]\!]$ returns the cost expression $c_{\mathtt{main}_1}$ computed in Example 3.1. □

4.3. The translation function

We have seen how the accumulated costs and their corresponding cost expressions can be computed for a given synchronization set. Interestingly, each synchronization set has its own accumulated cost, thus, for our analysis it is fundamental to keep the accumulated costs separate in order to get precise results. In the following section we define a `translate` function that computes the cost of a method considering all possible synchronization sets and synchronizations performed on it.

The translation of an `alt` method $\mathtt{m}$ consists of two steps: first (i) the synchronization schema $\mathcal{S}_{\mathtt{m}}$ is computed, following the technique described Section 4.1, then (ii) the body of method $\mathtt{m}$ is analyzed, by parsing each of its statements in order. The analysis (ii) records the accumulated costs of synchronization sets in *translation environments*, ranged over by Ψ, Ψ', $\cdots$. Translation environments are maps of the form $\mathcal{S}_{\mathtt{m}}(x) \mapsto \varepsilon$, which relate the different synchronization sets to their corresponding accumulated costs. The auxiliary operators $\parallel$ and $+$ are used as follows:

$$(\Psi \parallel exp)(S) = \Psi(\mathcal{S}) \parallel exp$$
$$(\Psi + exp)(\mathcal{S}) = \varepsilon \cdot \langle m(\overline{e}), exp' + exp\rangle \parallel (exp_1 + exp) \parallel \cdots \parallel (exp_n + exp)$$
$$\text{with } \Psi(S) = \varepsilon \cdot \langle m(\overline{e}), exp'\rangle \parallel exp_1 \parallel \cdots \parallel exp_n$$

Given a synchronization schema $\mathcal{S}$, the translation function, written as $\mathtt{translate}_{\mathcal{S}}(I, \Psi, x, l, t, s)$, takes the following six arguments:

$$\texttt{translate}_{\mathcal{S}}(I, \Psi, x, l, t, \mu) =$$

$$\begin{cases}
(1)\ (I, \Psi + e, l, t + e) & \text{when } \mu = \mathit{wait}(e) \\
(2)\ (I[f \mapsto \mathcal{S}(x)], \Psi, l + m(\overline{e}), t) & \text{when } \mu = \nu f\colon \mathtt{m}(y, \overline{z}, \overline{e}) \text{ and } y \in \mathcal{S}(x) \\
(3)\ (I[f \mapsto \mathcal{S}(y)], \Psi[\mathcal{S}(y) \mapsto \varepsilon \cdot \langle m(\overline{e}), 0\rangle], l, t) & \text{when } \mu = \nu f\colon \mathtt{m}(y, \overline{z}, \overline{e}) \text{ and } y \notin \mathcal{S}(x) \\
 & \varepsilon = \begin{cases} \Psi(\mathcal{S}(y)) & \text{if } \mathcal{S}(y) \in \mathrm{dom}(\Psi) \\ t & \text{otherwise} \end{cases} \\
(4)\ (I \setminus F, \Psi + l, 0, t + l) & \text{when } \mu = f^{\checkmark} \text{ and } x \in I(f) \\
 & \text{where } F = \{f' \mid I(f') = \mathcal{S}(x)\} \\
(5)\ (I \setminus F, (\Psi \parallel t') \setminus I(f), 0, t') & \text{when } \mu = f^{\checkmark} \text{ and } x \notin I(f) \\
 & \text{where } F = \{f' \mid I(f') = \mathcal{S}(x) \text{ or } I(f') = I(f)\} \\
 & \text{and } t' = \mathit{max}(t + l, [\![\Psi(I(f))]\!]) \\
(6)\ (I \setminus F, \Psi + l, 0, t + l) & \text{when } \mu = f^{\checkmark} \text{ and } f \notin \mathrm{dom}(I) \\
 & \text{where } F = \{f' \mid I(f') = \mathcal{S}(x)\}
\end{cases}$$

Fig. 3. The translation of `alt` statements.

i. I is a map from future names to synchronization sets,
ii. Ψ is a translation environment,
iii. x is the name of the *carrier*, i.e., the actor on which s is executing,
iv. l is a cost expression that over-approximates the cost of the methods invoked on actors belonging to the same synchronization set where the carrier x resides and not yet synchronized,
v. t is a cost expression that over-approximates the current computational cost, which is the computational time accumulated from the beginning of the method execution, and
vi. s is a sequence of statements, representing $\mathit{wait}(e)$, $\nu f\colon \mathtt{m}(\overline{e})$, or $f^{\checkmark}$ that must be translated.

and returns a tuple consisting of the four elements:

i. an updated map I',
ii. an updated translation environment Ψ',
iii. an updated the cost of methods running on actors in the same synchronization set as the carrier, and
iv. an expression of the updated current computational cost.

The function is defined on statements as follows

$$\begin{aligned}
\texttt{translate}_{\mathcal{S}}(I, \Psi, x, l, t, 0) &= (I, \Psi, l, t) \\
\texttt{translate}_{\mathcal{S}}(I, \Psi, x, l, t, \mu\,; s) &= \texttt{translate}_{\mathcal{S}}(I', \Psi', x, l', t', s) \\
 & \quad \text{where } \texttt{translate}_{\mathcal{S}}(I, \Psi, x, l, t, \mu) = (I', \Psi', l', t')
\end{aligned}$$

where $\texttt{translate}_{\mathcal{S}}(I, \Psi, x, l, t, \mu)$ is detailed in Fig. 3. Note that, the difficult case of the translation function is when the statement is a task synchronization $f^{\checkmark}$, which is covered by cases (4), (5) and (6). In the following, we discuss the six cases of Fig. 3 in detail.

***Case* (1):** When s is a $\mathit{wait}(e)$ statement, the `translate` function adds the cost e to the current cost t, which is also reflected on the environment Ψ by updating the accumulated cost of the method invocations on each synchronization set in Ψ.

***Cases* (2) *and* (3):** When s is a method invocation on actor y, there are two subcases, depending on whether y is in the same synchronization set of carrier x (*Case* (2)) or not (*Case* (3)). In *Case* (2), the cost of the invocation is added to l and the updated I binds the corresponding future f to $\mathcal{S}(x)$. In *Case* (3), we add the binding from f to $\mathcal{S}(y)$ to the map I and update the translation environment Ψ by appending the cost of invoking $\mathtt{m}$ to the accumulated cost to which $\mathcal{S}(y)$ maps. Note that, method invocations only includes the natural expressions used for invoking $\mathtt{m}$, that is, it includes $m(\overline{e})$.

***Case* (4):** This is the first sub-case of task synchronization which captures the situation where the synchronization is performed on a method whose callee belongs to $\mathcal{S}(x)$. Since it is non-deterministic when the task being synchronized will actually be scheduled, we add the sum of the costs of all the tasks running on actors in $\mathcal{S}(x)$, which is stored in l, to the current time t (worst case) at this synchronization point. The `translate` function then resets l to 0, and removes from I all the corresponding futures since the corresponding costs have been already considered.

$$3: \ \left(\emptyset,\ \emptyset,\ 0,\ t_3\right) \qquad [\, t_3 = k_0 \,]\ \textbf{(1)}$$

$$4: \ \left(\{f \mapsto \{x\}\},\ \emptyset,\ m_1, t_4\right) \qquad [\, t_4 = t_3 \,]\ \textbf{(2)}$$

$$5: \ \left(\{f \mapsto \{x\}, g \mapsto \{y,z\}\},\ \{\{y,z\} \mapsto t_4 \cdot \langle m_3, 0\rangle\},\ m_1,\ t_5\right) \qquad [\, t_5 = t_4 \,]\ \textbf{(3)}$$

$$6: \ \left(\{f \mapsto \{x\}, g \mapsto \{y,z\}\},\ \{\{y,z\} \mapsto t_4 \cdot \langle m_3, a\rangle\},\ m_1,\ t_6\right) \qquad [\, t_6 = t_5 + a \,]\ \textbf{(1)}$$

$$7: \ \left(\{f \mapsto \{x\}, g \mapsto \{y,z\}, h \mapsto \{w\}\}, \{\{y,z\} \mapsto t_4 \cdot \langle m_3, a\rangle, \{w\} \mapsto t_6 \cdot \langle m_4, 0\rangle\},\ m_1,\ t_7\right) \qquad [\, t_7 = t_6 \,]\ \textbf{(3)}$$

$$8: \ \left(\{f \mapsto \{x\}, g \mapsto \{y,z\}, h \mapsto \{w\}, i \mapsto \{y,z\}\}, \{\{y,z\} \mapsto t_4 \cdot \langle m_3, a\rangle \cdot \langle m_2, 0\rangle,\ \{w\} \mapsto t_6 \cdot \langle m_4, 0\rangle\},\ m_1,\ t_8\right) \qquad [\, t_8 = t_7 \,]\ \textbf{(3)}$$

$$9: \ \left(\{h \mapsto \{w\}\},\ \{\{w\} \mapsto (t_6 \cdot \langle m_4, 0\rangle \parallel t_9)\},\ 0,\ t_9\right) \qquad [\, t_9 = max\big(t_8 + m_1, t_4 + max(m_3, a) + max(m_2, 0)\big) \,]\ \textbf{(5)}$$

$$10: \ \left(\{h \mapsto \{w\}\},\ \{\{w\} \mapsto (t_6 \cdot \langle m_4, b\rangle \parallel t_9 + b)\},\ 0,\ t_{10}\right) \qquad [\, t_{10} = t_9 + b \,]\ \textbf{(1)}$$

$$11: \ \left(\emptyset,\ \emptyset,\ 0, t_{11}\right) \qquad [\, t_{11} = max\big(t_{10}, t_6 + max(m_4, b), t_9 + b\big) \,]\ \textbf{(5)}$$

$$12: \ \left(\emptyset,\ \emptyset,\ 0, t_{12}\right) \qquad [\, t_{12} = t_{11} \,]\ \textbf{(6)}$$

Fig. 4. Application of the `translate` function to *main* of Fig. 2. Every line l has pattern $l : (I_i, \Psi_i, l_i, t_i)\ [t_i]\ (c_i)$, where $[t_i]$ is the calculation of t_i and (c_i) is the translate function case applied.

***Case* (5):** The second sub-case describes the synchronization that is performed on a task whose callee, say y, does not belong to $\mathcal{S}(x)$. As actors x and y reside in different synchronization sets, the invocation on y is executed *in parallel* with the carrier x. Thus, the overall cost is computed as the *maximum* between the total cost of all pending invocations on the actors in $\mathcal{S}(x)$ to which carrier x belongs, captured by the local cost l, and the cost of all invocations on actors in $\mathcal{S}(y)$, denoted by $[\![\Psi(\mathcal{S}(y))]\!]$. Since at this point we have considered the worst scheduling, that is, we have already counted the cost of all methods spawned on the actors in $\mathcal{S}(y)$ and $\mathcal{S}(x)$ so far, we remove from the environment Ψ all invocations on the actors in $\mathcal{S}(y)$, as well as all the futures associated to $\mathcal{S}(y)$ and $\mathcal{S}(x)$ from I.

***Case* (6):** In this case, the future f does not belong to I, which implies that the cost of the invocation has been already computed. Nevertheless, it is possible that other methods have been invoked after this computation. Therefore, the actual termination of the invocation corresponding to f may happen *after* the completion of all the invocations. In order to take this into account, we need to add the cost of those tasks whose callee belongs to $\mathcal{S}(x)$, which has been accumulated in l, in a similar way as we have done in *Case (4)*.

We observe that the resulting translation environment Ψ_n is always empty because every method invocation is always synchronized within the method body. For the same reason, l'_n is always equal to 0. The following example is used to discuss in detail some relevant aspects of `translate`.

Example 4.2. Fig. 4 illustrates the application of `translate` function to `main` in Fig. 2. The leftmost column states the line of code i of `main` and we let I_i, Ψ_i, l_i, t_i refer to the corresponding argument of the function at line i. The second column contains the output tuple of `translate` function, while the third specifies the calculation of the current cost t_i. The rightmost column indicates the case (c_i) of `translate` function that we have applied. For the sake of clarity, as method calls do not contain numeric parameters, we use m for the cost expression $m(\_)$. Note that `translate` takes the synchronization schema $\mathcal{S}_{\texttt{main}} = \{\{x\}, \{y,z\}, \{w\}\}$ that has been computed in Section 4.1 as input.

At lines 3, 6 and 10, the cost expression t_i increases by summing the previous cost expression and the cost of *wait*$(\cdot)$-statements at each of these lines (case (1)). A local invocation (case (2)) to $\texttt{m}_1$ is added to l_4 at line 4 and its corresponding future variable is added to $I_4 = \{f \mapsto \{x\}\}$. Lines 5, 7 and 8 contain method invocations on actors not belonging to the carrier's synchronization set, which correspond to case (3) of the `translate` function. For instance, at line 5 we have the first method invocation on an actor in the synchronization set $\{y,z\}$. We then add the bindings $g \mapsto \{y,z\}$ to I_5 and $\{y,z\} \mapsto t_4 \cdot \langle m_3, 0\rangle$ to Ψ_5, where t_4 corresponds to the accumulated costs of methods invoked on the carrier's synchronization set so far, and the time distance to the subsequent method invocation on the same synchronization is 0. Observe that it is possible to compute the cost of pending invocations (on the actors of a synchronization set) by combining information in I and Ψ. In particular, I returns the synchronization set of the carrier of a future, Ψ returns the cost of the pending invocations on actors of that set.

The invocation of $\texttt{m}_4$ on actor w works analogously, where the associated synchronization set is $\{w\}$. As the call to $\texttt{m}_2$ at line 8 is invoked on z and $\mathcal{S}(y) = \{y,z\}$, we update Ψ_8 by appending the pair $\langle m_2, 0\rangle$ to $\Psi(\{y,z\})$, obtaining $\Psi_8(\{y,z\}) = t_4 \cdot \langle m_3, a\rangle \cdot \langle m_2, 0\rangle$. This accumulated cost expresses that the actors in $\{y,z\}$ have (*i*) two pending calls to be synchronized, namely $\texttt{m}_3$ and $\texttt{m}_2$; (*ii*) the time distance between these calls is a; and (*iii*) the first invocation on this synchronization set $\{y,z\}$ is performed at time t_4. Observe that the time distance a between $\texttt{m}_3$ and $\texttt{m}_2$ is set at line 6 by applying case (1) of `translate` function.

There are four synchronizations in this example, namely, $i^{\checkmark}$ at line 9, $h^{\checkmark}$ at line 11, and $f^{\checkmark}$, $g^{\checkmark}$ at line 12. Since $\Psi_8(\{y,z\}) = t_4 \cdot \langle m_3, a\rangle \cdot \langle m_2, 0\rangle$, the synchronization $i^{\checkmark}$ may have to wait for the termination of the calls to both $\mathtt{m}_2$ and $\mathtt{m}_3$. The function `translate` applies case (5) at line 9, which gives t_9 as the maximum between (*i*) the cost local to the carrier, that is, $t_8 + m_1$, where m_1 corresponds to the cost expression of $\mathtt{m}_1$ that is invoked on the carrier's synchronization set, which is pending to be synchronized and (*ii*) the expression $t_4 + max(m_3, a) + max(m_2, 0)$, which consists of three parts: (*a*) t_4 that is the starting time of the first call to an actor in $I(g) = \{y, z\}$, (*b*) the maximum between the cost of $\mathtt{m}_3$ and the distance a between the invocations of $\mathtt{m}_3$ and $\mathtt{m}_2$, and (*c*) the time needed to compute $\mathtt{m}_2$. Observe that $\{y, z\}$ is removed from Ψ_9, as the cost of the invocations on these two actors have already been calculated. Similarly, $I(i)$ and $I(g)$ are removed from I_9. Consequently, the synchronizations $f^{\checkmark}$ and $g^{\checkmark}$ at line 12 do not affect the cost expression. Observe also that t_9 is kept *in parallel* with the synchronization set $\{w\}$ in the translation environment, which allow us to calculate the correct costs of synchronizations in subsequent steps.

The application of case (1) at line 10 increases the cost local to the carrier to $t_{10} = t_9 + b$, and updates Ψ_{10} by adding b to the time distance in the pair $\langle m_4, b\rangle$ and to the parallel cost t_9. The synchronization $h^{\checkmark}$ at line 11 amounts to the maximal value of three elements: (*a*) t_{10} which is the local cost before the synchronization; (*b*) the cost expression $t_6 + max(m_4, b)$, which is the sum of the first invocation on an actor in $\{w\}$ and the cost of method $\mathtt{m}_4$; and (*c*) the cost $t_9 + b$ which is parallel to the method $\mathtt{m}_4$. Finally, the application of function `translate` at line 12 does not produce any effect. Thus, the cost of the `main` method in Fig. 2, $t_{\mathtt{main}}$ is the cost expressed by t_{12}, that is:

$$t_{main} = max(k_0 + a + max(b, k_4), max(k_0 + max(a, k_3 + k_4) + k_2, a + k_0 + k_1) + b) \qquad \square$$

5. Properties of the translation

The correctness of our system relies on the property that the computational time never increases during transitions. Therefore, the cost of the program in the initial configuration over-approximates the actual cost of every computation.

Cost programs The cost of a program is computed by solving a number of equations. Let a *cost program* be an equation system of the form

$$\begin{aligned} m_1(\overline{u}_1) &= exp_1 \\ &\vdots \\ m_h(\overline{u}_h) &= exp_h \\ main(\overline{u}) &= exp \end{aligned}$$

where m_i are function names, $\overline{u}_i$ natural formal parameters, and exp_i and exp are cost expressions. The *solution* of the above cost program is the closed-form upper bound for the equation $main(\overline{u})$, which is a function expressed in terms of its input parameters.

Definition 5.1 *(Cost of $\mathcal{P}$).* Let

$$\mathcal{P} = \left(\mathtt{m}_1(x_1, \overline{y}_1, \overline{u}_1) = \nu\overline{z}_1\,;s_1, \cdots, \mathtt{m}_h(x_h, \overline{x}_h, \overline{u}_h) = \nu\overline{z}_h\,;s_h, \mathtt{main}(x, \overline{u}) = \nu\overline{z}\,;s\right)$$

be an `alt` program. For every $1 \le i \le h$, let

1. $\mathcal{S}_{\mathtt{m}_i} = \mathtt{sschem}(\{\{x_i, \overline{y}_i\}\}, s_i)$,
2. $m_i(\overline{u}_i) = t_i$, where $\mathtt{translate}_{\mathcal{S}_{\mathtt{m}_i}}(\emptyset, \emptyset, x_i, 0, 0, s_i) = (I_i, \Psi_i, l_i, t_i)$,
3. $\mathcal{S} = \mathtt{sschem}(\{\emptyset\}, s)$ and $\mathtt{translate}_{\mathcal{S}}(\emptyset, \emptyset, x, 0, 0, s) = (I, \Psi, l_{main}, t_{main})$.

Let also $eq(\mathcal{P})$ be the cost program

$$(m_1(\overline{u}_1) = t_1, \cdots, m_h(\overline{u}_h) = t_h, main(\overline{u}) = t_{main})$$

A *cost solution* of $eq(\mathcal{P})$, named $\mathcal{U}(\mathcal{P})$, is the closed-form solution of the equation $main(\overline{u})$ in $eq(\mathcal{P})$.

For each method we produce cost equations which matches the cost of the method to the cost of its last statement, $m_h(\overline{u}_h) = t_h$. Analogously, we produce one extra equation for the cost of the `main` method $main(\overline{u})$ and its close-form solution over-approximates the computational time of an `alt` program.

Main result The correctness of our analysis follows by the statement below.

Theorem 5.1 *(Correctness). Let $\mathcal{P}$ be an `alt` program, whose initial configuration is cn, and $\mathcal{U}(\mathcal{P})$ be the closed-form solution of $eq(\mathcal{P})$. If $cn \Rightarrow^* cn'$, then $time(cn') \le \mathcal{U}(\mathcal{P})$.*

(1) `eq(m_pp(`$\overline{u}$`),e,[m_pp-1(`$\overline{u}$`))]),[])`
(2), (3) `eq(m_pp(`$\overline{u}$`),0,[m_pp-1(`$\overline{u}$`))]),[])`
(4), (6) `eq(m_pp(`$\overline{u}$`),`$numexp(l)$`,[calls(`l`),m_pp-1(`$\overline{u}$`)]),[])`
(5) `eq(m_pp(`$\overline{u}$`),0,[`$maxeq(pp, t+l, [\![\Psi(I(f))]\!])$`]),[])`

Fig. 5. PUBS cost equations production.

The proof of Theorem 5.1 is presented in the Appendix. It relies on (*i*) the extension of the function `translate` to run-time configurations, on (*ii*) defining the cost of a computation $cn \Rightarrow^* cn'$, noted $time(cn \Rightarrow^* cn')$ to be the sum of the labels of transitions, and on (*iii*) verifying that if $\mathcal{U}(\mathcal{P})$ is a solution of `translate`(cn) and $cn \Rightarrow^* cn'$ then $\mathcal{U}(\mathcal{P}) - time(cn \Rightarrow^* cn')$ is a solution of `translate`(cn').

As regards the extension in (*i*), we need to enhance the notion of synchronization set in order to refer to the current state of the objects. In this way, it is possible to define the cost equations of futures that have been already created (the corresponding task is either enqueued in a task pool, or running, or terminated). Overall, this enhancement requires the demonstration of a number of properties (see Lemma A.3, A.4, and A.5). In particular, Lemma A.5 shows that synchronization sets are stable over time. Namely, two different synchronization sets will never get merged during the execution. This is clearly very important as objects of two different synchronization sets are considered to execute in parallel in our analysis, while objects within the same synchronization set are considered to execute sequentially.

As regards (*iii*), we reason by induction on the length of $cn \Rightarrow^* cn'$ and we are reduced to two basic cases, according to the type of the last transition in $cn \Rightarrow^* cn'$:

1. either $cn'' \to cn'$; assuming that $\mathcal{U}$ is a solution of `translate`(cn''), then it is also a solution of `translate`(cn'), see Lemma A.15;
2. or $cn'' \stackrel{t}{\to} cn'$; assuming that $\mathcal{U}$ is a solution of `translate`(cn''), then $\mathcal{U} - t$ is also a solution of `translate`(cn'), see Lemma A.14.

The proof is complex because we have to deal with a lot of technical details, such as managing the synchronous sets at runtime and extracting the time difference in the cost equations when $cn'' \stackrel{t}{\to} cn'$ (which is not easy when costs are stored in Ψ).

It is important to observe that Theorem 5.1 is demonstrated using the (theoretical) solution of cost equations in [2]. This allows us to circumvent possible errors in implementations of the theory, such as `CoFloCo` [8] or PUBS [2]. As a byproduct of Theorem 5.1, we obtain the correctness of our technique, modulo the correctness of the solver.

6. The prototype

The analysis technique proposed in this paper has been prototyped using [1]. In this section, we are going to discuss a number of technical details about converting cost equations in Section 4 to an adequate format for PUBS [2], which is an equation solver. We also present a preliminary evaluation of our prototype by discussing a number of examples and compare our analysis with the parallel cost analysis in [4].

6.1. The conversion of cost equations

The `translate` function returns a set of equations that *need to be adapted* before inputting them in PUBS. In particular, the equations used by this tool have the form `eq(name,cost,calls,size)`, where `name` is the equation name, `cost` the value produced by this equation, `calls` a list of equations called from `name` and `size` the size relations needed to compute the closed form of the upper bound (see [2] for details). The reason why an adaptation is needed is that in one equation, PUBS add the cost of the expression `cost` plus the cost of all calls in `calls`, thus, using only one equation we cannot express the cost of $max(m_1, m_2)$, because the cost of methods $\mathtt{m}_1$ and $\mathtt{m}_2$ will be summed if we include m_1 and m_2 in `calls`. A maximum can be derived by generating two different equations with the same name, so that, as PUBS computes the worst cost, it applies the maximum between the two equations.

Let us explain how equations are generated by using the recursive execution of `translate`. Observe that, as it is detailed in Section 4.3, function `translate` is applied recursively to all statements in a method. Note also that the cost of each program can be expressed with respect to the cost of the previous program point with expressions like $t + e$, where t corresponds to the time computed for the previous program point. This can be seen in Example 4.2 where the cost produced by the application of `translate` to each statement can be expressed in terms of the cost produced by the previous executions with expressions like $t_6 = t_5 + a$ (see the right of Fig. 4). Then, for each program point, by means of the application of `translate`, we produce one cost equation capturing how the time advances, that is, how t is modified in the different cases of Fig. 1.

Fig. 5 illustrates how the execution $\mathtt{translate}_{\mathcal{S}}(I, \Psi, x, l, t, pp{:}\mu)$, where pp corresponds to the program point of instruction μ, produces its corresponding equation for the different cases of Fig. 3 applied to method `m`. We use `m_pp` to refer to the new equation produced at program point i of method `m`, which captures the cost at this point, that is t_i

```
eq(main(A,B),0,[main_12(A,B)],[]).

eq(main_2(A,B),0,[],[]).
eq(main_3(A,B),c(k0),[main_2(A,B)],[]).
eq(main_4(A,B),0,[main_3(A,B)],[]).
eq(main_5(A,B),0,[main_4(A,B)],[]).
eq(main_6(A,B),nat(A),[main_5(A,B)],[]).
eq(main_7(A,B),0,[main_6(A,B)],[]).
eq(main_8(A,B),0,[main_7(A,B)],[]).

eq(main_9(A,B),0,[main_9_max(A,B)],[]).
eq(main_9_max(A,B),0,[main_8(A,B),m1(A)],[]).
eq(main_9_max(A,B),0,[main_4(A,B),main_9_1_max(A,B), main_9_2_max(A,B)],[]).

eq(main_9_1_max(A,B),0,[m3(A)],[]).
eq(main_9_1_max(A,B),nat(A),[],[]).

eq(main_9_2_max(A,B),0,[m2(A)],[]).
eq(main_9_2_max(A,B),0,[],[]).

eq(main_10(A,B),nat(B),[main_9(A,B)],[]).

eq(main_11(A,B),0,[main_11_max(A,B)],[]).
eq(main_11_max(A,B),0,[main_10(A,B)],[]).
eq(main_11_max(A,B),0,[main_6(A,B),main_11_1_max(A,B)],[]).
eq(main_11_max(A,B),nat(B),[main_9(A,B)],[]).

eq(main_11_1_max(A,B),0,[m4(A)],[]).
eq(main_11_1_max(A,B),nat(B),[],[]).

eq(main_12(A,B),0,[main_11(A,B)],[]).

eq(m1(A),c(k1),[],[]).
eq(m2(A),c(k2),[],[]).
eq(m3(A),c(k3),[m4(A)],[]).
eq(m4(A),c(k4),[],[]).
```

Fig. 6. Equations from the `translate` function Fig. 4.

in Fig. 4. We also use `m_pp-1` for the equation of the preceding program point. For case (1) the equation adds the cost expression e with the cost of the previous program point `m_pp-1`. Cases (2) and (3) only adds the cost of the previous program point. Cases (4) and (6) adds the cost of l, which might contain natural expressions, like constants or natural names, or can contain calls to other methods or to previous program point equations. We define two functions to handle these two cases: $numexp(e)$, which returns the addition of the natural expressions in e; and $calls(e)$, which returns a sequence of all non-natural expressions in e. Thus, $numexp(l)$ are added to the cost expression of the equation, and calls to other equations, captured by $calls(l)$, are added to the `calls` of the equation, including also the cost of the previous program point `m_pp-1`$(\overline{u})$. The following example illustrates the production of the equations according to the application of `translate` function shown in Fig. 4.

Example 6.1. Fig. 6 outputs the equations corresponding to `translate` function shown in Fig. 4. The solver uses $nat(e)$ to refer to $max(e, 0)$ and to avoid negative input values (see equations `main_6` and `main_10`), and $c(k)$ for constant values (see equations `main_3`, `m1`, `m2`, `m3` and `m4`). Note that the cost expressions are composed step by step by referencing the previous cost equation in the calls of a cost equation, as it is shown in all cost equations. New cost is added according to case (1) e.g., `main_3`, which sums the cost of `main_2` and `c(k0)`, or `main_6`, which sums the cost of `main_5` and `nat(A)`. For instance, expressions like $t_6 = t_5 + a$ are captured by the equation

```
eq(main_6(A,B),nat(A),[main_5(A,B)],[]).
```

□

The computation of the accumulated costs in case (5), which include *max* expressions, requires a specific treatment. To deal with expressions of the form $max(exp_1, \ldots, exp_n)$, we produce n equations of the forms $maxeq(\overline{u}) = exp_1; \ldots; maxeq(\overline{u}) = exp_n$. *max* expressions are processed by the function $maxeq(id, exp_1, \ldots, exp_n)$ which generates the following n equations for the identifier id:

Program	t_a	t_s	#eq	Upper bound
NWorkers (t1,t2,lt)	1	194	119	`max(t2,lt,t1)`
NWorkersMultiWorks (t1,t2,t3,t4)	5	591	231	`t1+t2+t3+t4`
NWorkersPartSync (t1,t2,lt)	1	73	57	`lt+max(lt+lt,t2,lt+max(t1,lt))`
NWorkersDelayed (t1,t2,t3,t4,lt)	1	237	87	`t1+max(lt,t2)+t3+t4`
ProdCons (pt,ct)	1	132	118	`max(ct+max(pt+pt,ct+pt))`
MapReduce (mt1,mt2,lt1,rt1,rt2,lt2)	2	3665	1330	`lt1+max(mt2,mt1)+lt2+max(rt1,rt2)`
MapRedSubWorkers(mt,rt,sm1,sm2,sr1,sr2)	3	11059	1378	`mt+max(sm1,sm2)+rt+max(sr1,sr2)`

Fig. 7. Experimental results (times in ms).

$$\begin{array}{c} \texttt{eq(m\_id\_max(}\overline{u}\texttt{)},\ natexp(exp_1),\ \texttt{[}calls(exp_1),\ maxexp(exp_1)\texttt{]},\ \texttt{[]}\texttt{)} \\ \vdots \\ \texttt{eq(m\_id\_max(}\overline{u}\texttt{)},\ natexp(exp_n),\ \texttt{[}calls(exp_n),\ maxexp(exp_n)\texttt{]},\ \texttt{[]}\texttt{)} \end{array}$$

As before, in the equations we distinguish the cost corresponding to natural values and the cost associated to calls. In addition, we use *maxexp*(*exp*) to separate the cost associated to other *max* expressions that could be found in the accumulated cost. For each *max* expression, we use to *maxeq* function with a fresh identifier for processing other possible *max* expressions found within the expressions and apply it recursively.

Example 6.2. For instance, let us produce the equations for the cost expression

$$t_9 = max(t_8 + m_1, t_4 + max(m_3, a) + max(m_2, 0))$$

at line 9 in Fig. 4 is captured by the two `main_9_max` equations in Fig. 6, which respectively includes the cost `main_8` + `m1`, and the cost of `main_4 + main_9_1_max + main_9_2_max`. Analogously, `main_9_1_max` and `main_9_2_max` corresponds to the maximums $max(m_3, a)$ and $max(m_2, 0)$, respectively. Similarly, three equations `main_11_max` are needed to compute *max* expression of t_{11} at line 11 in Fig. 4. By solving the equations shown in Fig. 6 with `PUBS`, we get the expression:

$$\begin{array}{l} max([nat(A) + c(k_0) + max([nat(B), c(k_4)]), \\ \quad nat(B) + max([c(k_0) + max([nat(A), c(k_3) + c(k_4)]) + c(k_2), \\ \qquad\qquad nat(A) + c(k_0) + c(k_1)])]) \end{array}$$

which reflects the cost expression t_{main} computed in Fig. 4 and captures the overall cost of `main` in Example 4.2. □

Another point to remark is that PUBS is able to detect non-terminating programs caused by direct or indirect recursions, indicating the solution of the equations is unbounded.

6.2. Experimental evaluation

The prototype can be experimented online.[1] To deliver preliminary assessments, we have written our programs in a language that is an extension of `alt` – the `ABS` language [13], where only the features presents in `alt` are used. The experiments, whose source codes are available in the tool web site, model a number of typical concurrent and distributed scenarios:

NWorkers(t1,t2,lt) models the standard *fork-join* pattern where multiple workers execute in parallel on different actors parts of a larger task, represented by *wait*(*t1*) and *wait*(*t2*), respectively, and the carrier spends `lt` time units in parallel to the workers.

NWorkersMultiWorks(t1,t2,t3,t4) models a similar scenario but spawning four tasks per worker, with durations t1, t2, t3, t4 and synchronizes all of them at the end of the main method.

NWorkersPartSync(t1,t2,lt) models a *fork-join* pattern with two workers, but the launcher spends lt time units, between the synchronizations;

NWorkersDelayed(t1,t2,t3,t4,lt) also models a *fork-join* pattern with two workers that execute four tasks with durations t1, t2, t3, t4. In this model, the launcher spends lt time units between spawning the second and the third task.

ProdCons(pt,ct) models a *producer-consumer* scenario where the production and the consumption might occur in parallel, taking pt and ct time units for producing and consuming, respectively.

[1] http://costa.ls.fi.upm.es/timeanalysis.

MapReduce(mt1,mt2,lt1,rt1,rt2,lt2) models the *map-reduce* algorithm where the *map* tasks, which take mt1 or mt2, are spawned on multiple workers in parallel. When all *map* tasks are finished, after lt1 time units for modeling the *map* result processing, the *reduce* tasks, which take rt1 or rt2, are spawned on the workers. Then after the synchronizations, the main methods spends lt2 time units to process the results.

MapReduceSubWorkers(mt,rt,sm1,sm2,sr1,sr2) models the *map-reduce* algorithm where the workers share the task, taking times mt for *map* and rt for *reduce*, and launch subtasks with times sm1, sm2 for *map* and sr1, sr2 for *reduce* to other sub-workers.

We have executed the above codes on an Intel Core i7-7500U CPU @ 2.70 GHz with 64 Gb of RAM, running Debian 9.3. Table 7 summarizes the upper bounds obtained by means of our prototype. For the sake of clarity, we have simplified the expressions that are redundant, and we have replaced in the table `nat(e)` by `e`. For instance, the expression returned for NWorkers, namely `max([nat(t2),nat(lt),nat(t1),nat(lt)])`, is written as `max([t2,lt,t1])`.

The results of Table 7 show that the computational time of our analysis is quite short: less than 5 ms for all the programs. The number of equations might be high for big programs, as it can be seen in MapReduce or in MapReduceSubWorkers. Even so, the time required to solve them is higher but is still rather short. This number of equations can be reduced by producing only one equation for those sequence of instructions that do not modify the time, reducing the time taken in solving them.

Regarding to the precision of our analysis, the results highlight that we do not lose precision in *fork-join* patterns. In fact, in this case, we properly identify those parts that can execute in parallel and use the `max` to capture the parallelism. For instance, in NWorkers and NWorkersMultiWorks, our tool identifies the parts running in parallel and calculates the cost as the maximum cost needed among the workers, independent from the number of workers executing simultaneously (in this programs we have four different workers). Similarly, as highlighted by the programs MapReduce and MapReduceSubWorkers, our tool identifies that *map* and *reduce* tasks might execute in parallel before their synchronization. In this case, their costs are added only once to the total cost inside the two maximum expressions `max(mt2,mt1)` and `max(rt2,rt1)`.

6.3. Tools comparison

In the following, we compare the results obtained by our approach and by the analysis described in [4].

The analysis of [4] uses a language – the `ABS` language [13] – that is actually a superset of `alt`. In particular, in order to solve the syntactic mismatches, we had to modify the *wait*(*e*)-statement into loops with *e* iterations so as to model the time progress. Additionally, we ignore the constant times output by [4] as they are not considered by our time analysis.

The table illustrates the results obtained by applying our time analysis and the analysis of [4] to the `alt` programs in Examples 3.1, 3.2 and 3.3.

Prog.	Time Analysis	Analysis of [4]
$main_1$	k_0+max(a,k_1)+max(b,k_2)	max(k_0+a+b,k_0+k_1+k_2+a)
$main_2$	k_0+max(a+b+k_2,k_1)	k_0+max(k_1+k_2,a+b+k_2)
$main_3$	k_0+max(a,k_2+k_3)+max(b,k_1)	k_0+max(k_0+a+b,k_0+k_1+k_2+k_3,k_0+a+k_1+k_2)

To explain this point, we need to recall the technique of [4], which uses *block-level cost-centers* to compute the cost expression of the *sequential parts* of the program. A *distributed flow graph* of the program is then generated to express tasks that might be executed in parallel and their flow relations. The distributed flow graph is used to find all the paths that start from the initial node and end in any possible final node. The cost expressions of every path are computed accordingly and the overall cost, called the *parallel cost* of the program, is simply the maximum of the costs. The critical point of the technique in [4] is that, in order to take into account every possible scheduler's choice, the distributed flow graph collects edges that connect the beginning and the end of those methods that might be simultaneously in the pool of pending tasks. These edges introduce cycles in the graph; hence, the collection also includes those paths where parallel methods are sequentialized. This results in a loss of precision.

The reader may notice that the results obtained in some scenarios by our time analysis are more precise than [4] for the studied programs. For instance, this precision loss in [4] is shown in $main_1$, where the cost of `a` and k_1 are not considered to happen in parallel. Similarly, in $main_2$, where the costs k_1 and k_2 are summed; or in $main_3$, where k_2 should not be added to the expression k_0+a+k_1+k_2. As the values of the upper bound depend on the input values, cost expressions cannot be directly compared. In order to compare the results obtained by both tools, we evaluate the upper bound for all possible combinations of the input arguments, including constant values, ranging each of them between [1-100] and compare the results obtained with both tools. The following table summarizes the results obtained by showing the percentage of evaluations where our approach is more precise ($\%_w$), less precise ($\%_l$) or equally precise ($\%_e$) with respect to the technique in [4]:

Program	($\%_w$)	($\%_l$)	($\%_e$)
$\texttt{main}_1$	75.00%	12.09%	12.91%
$\texttt{main}_2$	16.17%	0.00%	83.83%
$\texttt{main}_3$	34.62%	34.62%	30.74%

The results of the evaluations show that our technique obtains better results than [4] for programs for $\texttt{main}_1$ and $\texttt{main}_2$, and for $\texttt{main}_3$ we get better results for 34% of the evaluations and worse for the same number of evaluations.

Regarding the benchmarks of Fig. 7, the analysis of [4] gives results similar to our tool, except for NWorkersDelayed. In fact, this program is the only one spends some time between two invocations on the same actor. For this program, the upper bound expression obtained by [4] is `lt+t1+t2+t3+t4`, which sequentialize the cost of all the methods. On the contrary, our tool is able to detect that `lt` and `t2` are costs of two parallel methods. Thus, our analysis does not sequentialized these two costs, but returns their maximal value, which in turn produces the expression `t1+max(lt,t2)+t3+t4`.

7. Related work

Static time analysis techniques for concurrent programs follow two main approaches: those based on type-and-effect systems and those based on abstract interpretation.

Type-and-effect systems [9] (i) collect constraints on type and resource variables and (ii) solve these constraints by means of an off-the-shelf solver [2,8]. Recent work has applied type-based amortized analysis for deriving upper bounds of parallel first-order functional programs [12]. This work differs from our approach in the concurrency model, as they do not handle references to actors in the programs and there is no distinction between blocking and non-blocking synchronization.

In this paper we do not perform constraint collection on type and resources. This is because `alt` is intended to be a behavioral type language; therefore, we directly use `alt` to define a (compositional) analysis that returns cost equations. The main difference with respect to the analysis of [9] is that now we are able to compute the cost of functions that contain invocations on arguments, namely on actors that are already alive before the function invocations. An approach similar to our one has been proposed in [15] for verifying safety properties of sequential languages.

Abstract interpretation techniques addressing domains carrying quantitative information, such as resource consumption, have been proposed in the literature – see references in [16]. Consequently, several well-developed automatic solvers for cost analysis already exist. These solvers either use finite domains or use expedients (widening or narrowing functions) to guarantee the termination of the fix-point generation. For this reason, solvers often return inaccurate answers when fed with systems that are finite but not statically bounded. Among the others, [4] defines a technique based on abstract interpretation that targets a language with a similar concurrency model as presented in this paper. We have discussed the technique of [4] in Section 6, where we have also analyzed the differences with our technique. We recall that our technique returns more precise costs for programs that spawn several invocations without synchronization on the same synchronization set. In particular, [4] manifests a sensible loss of precision when cycles appear in the distributed flow graph, as all nodes in the cycle will be part of the path that leads to the maximum upper bound. We also observe that, since the technique in [4] is not compositional, it does not require any management of synchronization sets, which entangles a lot our technique.

8. Conclusions

This paper presents a technique for computing the time of concurrent programs. We have defined `alt`, an actor calculus that is intended to be a compilation target for concurrent languages featuring actor creation, task invocation and synchronization. In particular, `alt` features an explicit cost annotation that defines the number of machine cycles required before executing the continuation, which abstracts away the actual computation activities of the program. The computational time is then measured by introducing the notion of (strong) *t-stability* (*cf.* Definition 2.1), which represents the ticking of time and expresses that no control activity is possible up to t time units. In order to relate actors that might potentially delay executions of other actors, we introduce the notion of synchronization sets. Then, we define a `translate` function that uses synchronization sets to compute a cost equation function for each method definition. We have also proven that our approach is sound with respect to the actual computational time (*cf.* Theorem 5.1). The analysis has been prototyped and experimented, which shows that our approach produces accurate over-approximation.

Overall, our technique allows one to estimate the computational time of the source program by computing the cost of the target actor program. The aim is to use our technique in a cloud computing setting because `alt` terms might be considered as abstract descriptions of services and the estimation of the computational time may be used for enforcing the compliance with the service level agreement contract. In this context, the cost expressions in $wait(\cdot)$-statement, might be defined by means of a worst-case execution time analysis [5].

Future lines of work must consider the possibility of obtaining more precise information about *synchronization sets* of actors. In this paper, these sets are computed for method bodies by simply accumulating information, despite of the fact that two actors, in a stage of the computation may affect each other, while in another stage they are independent. In fact, a more appropriate notion would have been that of *sequence of synchronization sets*. While such improvement will end up in more precise results of the cost analysis, it is not clear how much more complex the theoretical developments will become. Perhaps a manageable extension will be a notion in between the synchronization sets and the sequences of such sets.

In this paper, the cost of a method includes that of the asynchronous invocations in its body because every invocation is synchronized. This constraint does not permit to trigger methods such as drivers, daemons, etc. without waiting for their termination. In order to take care of costs of unsynchronized methods, one should use *continuations* and compose costs according to the synchronization set they correspond. We will address this extension in a future work.

Appendix A

We introduce this appendix with some helpful notation related to the cost equation of an `alt` program, and with few algebraic notation on graphs and equivalence relations that will be helpful to define synchronization schema for runtime configurations.

Definition A.1. In this appendix, to simplify future notations, we extend the notion of cost expression with future names, to be able to refer to the cost of currently running processes. We also extend the *max* operator to take a variable number of parameters (with the standard semantics), and thus have the following syntax definition:

$$exp \ ::= k \ \mid \ u \ \mid \ f \ \mid \ exp + exp \ \mid \ \mathtt{m}(\overline{e}) \ \mid \ max(\overline{exp})$$

Additionally, we extend the syntax of accumulated costs as follows, to allow to count the cost of running process as well as functions yet to be executed:

$$\varepsilon \ ::= exp \ \mid \ \varepsilon \cdot \langle m(\overline{e}), exp \rangle \ \mid \ \varepsilon \cdot \langle f, exp \rangle \ \mid \ \varepsilon \parallel exp \ .$$

The parameters $\mathcal{S}$, I, Ψ, x, l, t and s of the `translate` function are *consistent* iff: $\mathcal{S}$ is an equivalence relation on a set including all free actor names in s and x; all free future names in s are in $\mathrm{dom}(I)$; all the synchronization sets in $\mathrm{im}(I)$ are in $\mathcal{S}$;[2] and all the synchronization sets in $\mathrm{im}(I)$, minus $\mathcal{S}(x)$, are in $\mathrm{dom}(\Psi)$.

Suppose given consistent parameters $\mathcal{S}$, I, Ψ, x, l, t and s. We write $\mathtt{T}_{\mathcal{S}}(I, \Psi, x, l, t, s)$ the cost of the translation of s:

$$\mathtt{translate}_{\mathcal{S}}(I, \Psi, x, l, t, s) = (I', \Psi', l', t') \Leftrightarrow \mathtt{T}_{\mathcal{S}}(I, \Psi, x, l, t, s) = t'$$

Definition A.2. Suppose given any two cost expressions exp_1 and exp_2. We write $exp_1 \leq exp_2$ iff for all substitution Σ with $fv(exp_1) \cup fv(exp_2) \subseteq \mathrm{dom}(\Sigma)$ and $\mathrm{im}(\Sigma) \subseteq [0..\infty]$, we have $\Sigma(exp_1) \leq \Sigma(exp_2)$.

The following definition is a small adaptation of Definition 5.1 where the cost of the main statement of the program, instead of being by itself, is mapped to a name f_{start}. That way, a cost program becomes a simple set of equations, that is easier to manipulate in the proofs (see Lemma A.13).

Definition A.3. Suppose given a set of functions $\overline{FD} = \mathtt{m}_1(\overline{x}_1) = s_1, \cdots, \mathtt{m}_n(\overline{x}_n) = s_n$. The *cost program eq* of an `alt` program $P = (\overline{FD}, s_{main})$ is defined as follows:

$$\begin{aligned} eq(P) \stackrel{def}{=} & [f_{start} \mapsto \mathtt{T}_{\mathtt{sschem}(\{\ \{start\}\ \}, s_{main})}(\emptyset, \emptyset, start, 0, 0, s_{main})] \\ & \cup \bigcup_i [\mathtt{m}_i(\overline{x}_i) \mapsto \mathtt{translate}(\mathtt{m}_i(\overline{x}_i) = s_i)] \end{aligned}$$

A *cost solution* Σ of an `alt` program P is a solution for the equation $eq(P)$.

Finally, the following definition introduces some notations on graphs and relations. They will be useful in the next section, where we define the notion of synchronization schema on runtime configurations.

Definition A.4. Given a set V and a relation R on V, we write $R^{eq(V)}$ the transitive, reflexive and symmetric closure of R in V. Note that we will write R^{eq} when the set V is clear from the context. Given a set V and an equivalence relation R on V, we write $\tilde{R}$ the partition of V raised by R. Given a graph $G = (V, E)$ and an equivalence relation R on V, we write $G/^{*}R$ the graph $G' = (V', E')$ such that:

$$V' \stackrel{def}{=} \tilde{R} \qquad E' \stackrel{def}{=} \{(A, A') \mid A \neq A' \in R, \exists x \in A, y \in A', \ (x, y) \in E\}$$

In the rest of the appendix, we write: $parents(G, x)$ for the set of the parent nodes of x in G, i.e., $parents(G, x) \stackrel{def}{=} \{y \mid (y, x) \in E\}$; $paths(G)$ for the set of non-empty paths p in the directed graph G; $start(p)$ for the start node of the path p; $end(p)$ for the end node of the path p; and $nodes(p)$ for the set of nodes traversed by the path p.

[2] We note $\mathrm{im}(I)$ the image of I, i.e., $\mathrm{im}(I) = \{I(f) \mid f \in \mathrm{dom}(I)\}$.

A.1. Runtime analysis

In this section, we extend the translation function of Fig. 3 to the runtime terms of `alt`. This extension maps all tasks in the runtime to a cost equation encoding an upper bound of its finishing time. In order to do so, we need to define the different parameters of the `translate` function. First of all, let us introduce the necessary notations:

Definition A.5. Suppose given a runtime configuration cn. We write $x \in cn$ iff there exists p, q and cn' such that $cn = act(x,p,q)\ cn'$. If $cn = act(x,p,q)\ cn'$, we write $cn(x)$ for the configuration $act(x,p,q)$. If $cn = act(x,p,q)\ cn'$, we write $(x:f) \in cn$ iff there exists s; $f \in p\ q$ or if $cn' = invoc(x,f,\mathtt{m},\overline{v})\ cn''$; we write $f \to f' \in cn$ iff there exists s; $f \in p\ q$ and $f'^{\checkmark}$ is a statement in s; and we write $f \to x' \in cn$ iff there exists s; $f \in p\ q$ such that x' is free in s or if there exists f' such that $f \to f' \in cn$ and $(x':f') \in cn$. Finally, we write $\mathtt{roots}(cn)$ for the set of root futures of cn, i.e., $\mathtt{roots}(cn) = \{f \mid \nexists f',\ f' \to f \in cn\}$; and $A(cn)$ for a set of actor names A, which extracts from cn all actors named in A, with their futures $fut(f,val)$ and invocation messages $invoc(x,f,\mathtt{m},\overline{v})$.

Synchronization schema The first parameter, the synchronization schema, has a central role in the cost computation, as it captures the actors that may reciprocally affect the queues of each other by means of function invocations and synchronizations. Without such information, the cost computation cannot capture all the tasks that could occur in an actor's process queue, and thus would be erroneous. At static time, such schemas are computed using the parameters of method calls. However, at runtime, this information is missing as running code does not refer back to their originating method and parameters. The way we can recreate information captured by a synchronization schema at runtime is to follow the point-to relation between objects. We recall the point-to relation between object in the following definition:

Definition A.6. The *point-to* relation of a runtime configuration cn, noted $point\text{-}to(cn)$, is a directed graph (V,E) where $V \stackrel{def}{=} \{x,f \mid (x:f) \in cn\}$; $E \stackrel{def}{=} \{(x,f) \mid (x:f) \in cn\} \cup \{(f,x) \mid f \to x \in cn\}$.

With this, we can construct the synchronization schema of a runtime configuration. Following Definition 4.1, synchronization sets actually capture a notion slightly more subtle than simply affecting each other's queue. Firstly, synchronization sets do not include the obvious parent-son object relation, as this would be a too restrictive approximation that would consider any concurrent computation as put in sequence. Secondly, synchronization sets put in relation not only objects that may influence each other, but also objects that refer to influencing objects: the synchronization set somehow consider the transitive closure of the point-to relation. Finally, we also need to include in the synchronization sets the dependencies that might be created by future method calls. These considerations lead to the following definition of the synchronization sets for runtime configuration:

Definition A.7. Give a runtime configuration cn and a graph $G = (V,E)$ where the vertexes are sets of object and futures names in cn, we define the relation I on V as follows:

$$v\ I\ v' \Leftrightarrow x \in v, f \in v', (x:f) \in cn$$

This relation I is used to identify an object with its processes in the point-to relation.

Given a graph $G = (V,E)$, we define the relations P and L on V as follows:

$$x\ P\ y \Leftrightarrow \{y\} \subsetneq parents(G,x)$$

$$x\ L\ y \Leftrightarrow \exists p \in paths(G),\ start(p) = end(p) \wedge \{x,y\} \in nodes(p)$$

The relation P corresponds to the fact that an object x with more than one parent y must be in the same synchronization set; the relation L corresponds to the fact that objects in a dependency loop are in the same synchronization set.

The *synchronization graph* of a runtime configuration cn, written G_{cn} is defined as follows:

$$G_{cn} \stackrel{def}{=} (\{\mathrm{flat}(S) \mid S \in V\}, \{(\mathrm{flat}(S), \mathrm{flat}(S')) \mid (S,S') \in E\})$$

$$\text{where } \begin{cases} (V,E) = point\text{-}to(cn)/\overset{*}{P^{eq}}/\overset{*}{I^{eq}}/\overset{*}{L^{eq}} \\ \mathrm{flat}(M) = \bigcup_{N \in M} \bigcup_{O \in N} O \cap \{x \mid x \in cn\} \end{cases}$$

The *synchronization schema* of a runtime configuration cn, written $\mathcal{S}_{cn}$ is the set of vertices of G_{cn}.

Note that, by construction, $\mathcal{S}_{cn}$ is a partition of the object names in cn. Hence, we can use the same notation as in Section 4 by writing $\mathcal{S}_{cn}(x)$ for the set $A \in \mathcal{S}_{cn}$ containing x.

Future localization The parameter I is the simplest to define, now that we have a precise definition of synchronization schema. Indeed, this parameter simply maps the futures living in the configuration to its actor's synchronization set. Given a runtime configuration cn, we define:

$$I_{cn} \stackrel{def}{=} [f \mapsto \mathcal{S}_{cn}(x) \mid x \in cn,\ f \in cn(x)]$$

$$\texttt{translate}_{\mathcal{S}}(I, cn) = \begin{cases} (1) & \emptyset & \text{when } cn = \varepsilon \\ (2.1) & \emptyset & \text{when } cn = \mathit{fut}(f, \bot) \\ (2.2) & \{f \mapsto 0\} & \text{when } cn = \mathit{fut}(f, \top) \\ (3) & \bigcup_{p' \in p,q} \texttt{translate}_{\mathcal{S}}(I, x, p') & \text{when } cn = \mathit{act}(x, p, q) \\ (4) & \{f \mapsto \mathtt{m}(x, \overline{v})\} & \text{when } cn = \mathit{invoc}(x, f, \mathtt{m}, \overline{v}) \\ (5) & \texttt{translate}_{\mathcal{S}}(I, cn_1) \cup \texttt{translate}_{\mathcal{S}}(I, cn_2) & \text{when } cn = cn_1\ cn_2 \end{cases}$$

$$\texttt{translate}_{\mathcal{S}}(I, x, p) = \begin{cases} (6) & \emptyset & \text{when } p = 0 \\ (7) & \{f \mapsto \mathtt{T}_{\mathcal{S}'}(I', \Psi, x, l, 0, s)\} & \text{when } p = s\,;f \text{ and with } [\mathcal{S}, I]^x_s = (\mathcal{S}', I', \Psi, l) \\ (8) & \{f \mapsto 0\} & \text{when } p = f \end{cases}$$

Fig. 8. The translation of `alt` runtime terms.

Translation environments The `translate` function has a last important parameter: the environment Ψ giving the accumulated cost of every running synchronization set. At runtime, waiting statements are not only bound to numerical expression or method calls, but partially to executed processes identified by futures. This motivates the expression extension we gave in the introduction of the appendix. During the static time analysis, and also in the runtime analysis, Ψ contains an entry for all synchronization sets that have a non-empty process queue. Hence, this parameter is relevant only when translating a process, and collects data about the futures and synchronization sets related to the analyzed process.

The translation function We present in Fig. 8 the translation function of runtime configurations into mappings from future names to cost equations. Note that when translating a specific process, we need to use a local future localizations I, which takes into account objects that are not created yet, and to create the corresponding translation environment Ψ. This is done with the following function:

$$[\mathcal{S}, I]^x_s \stackrel{def}{=} (\mathcal{S}', I', \Psi, l) \quad \text{with} \quad \begin{cases} \mathcal{S}' = \mathtt{sschem}(\mathcal{S}, s) \\ I' = [f \mapsto A \mid f \in \mathrm{dom}(I),\ I(f) \subset A,\ A \in \mathcal{S}'] \\ \Psi = [A \mapsto \langle \sum_{I(f)=A,\ f \in fv(s)} f, 0\rangle \mid A \in I'(fv(s)) \wedge x \notin A] \\ l = \sum_{\substack{f \in I^{-1}(\mathcal{S}(x)) \\ f \text{ free in } s}} f \end{cases}$$

Definition A.8. Suppose given a set of functions $\overline{FD} = \mathtt{m}_1(\overline{x}_1) = s_1, \cdots, \mathtt{m}_n(\overline{x}_n) = s_n$. The *cost configuration eq* of a runtime configuration *cn* based on the functions $\overline{FD}$, written $eq(cn, \overline{FD})$, is defined as follows:

$$eq(cn, \overline{FD}) \stackrel{def}{=} \texttt{translate}_{\mathcal{S}_{cn}}(I_{cn}, \Psi_{cn}, cn) \cup \bigcup_i [\mathtt{m}_i(\overline{x}_i) \mapsto \texttt{translate}(\mathtt{m}_i(\overline{x}_i) = s_i)]$$

A *cost solution* Σ of a runtime configuration *cn* based on the functions $\overline{FD}$, is a solution for the cost configuration $eq(cn, \overline{FD})$.

The following two lemmas give some properties about the image of a cost equation, so we will be able to manipulate them in the rest of the proofs. The first lemma states that a cost equation maps functions and futures to expressions exp_x.

Lemma A.1. *Consider a consistent set of parameters $\mathcal{S}$, I, Ψ, x, l, t and s. Then, $\mathtt{T}_{\mathcal{S}}(I, \Psi, x, l, t, s)$ is a cost expression. Consider additionally a runtime configuration cn and a set of functions $\overline{FD}$. Then, the image of $eq(cn, \overline{FD})$ is a set of cost expressions exp.*

Proof. Inspecting the rules in Fig. 3, it is clear that t' in $(I, \Psi, l', t') = \texttt{translate}_{\mathcal{S}}(I, \Psi, x, l, et, s)$ is a cost expression. Consequently, by Definition A.1, $\mathtt{T}_{\mathcal{S}}(I, \Psi, x, l, t, s)$ is a cost expression as well. Additionally, it is easy, looking at the rules in Fig. 8, and by Definition A.8, to see that the image of $eq(cn, \overline{FD})$ are all constructed from $\mathtt{T}_{\mathcal{S}}(I, \Psi, x, l, 0, s)$ where l is a sum over futures. Hence, the image of $eq(cn, \overline{FD})$ is a set of cost expressions *exp* (a sum of future is a valid cost expression). □

Lemma A.2. *A cost expression exp is always equal to some $max(exp_1, \ldots, exp_n)$ where none of the exp_i contain a max operator.*

Proof. This property is a consequence of the following equations that can move all the *max* functions in e to the top level:

$$max(exp_1, \ldots, exp_n) + exp = max(exp_1 + exp, \ldots, exp_n + exp)$$
$$max(max(\overline{exp}), \overline{exp}') = max(\overline{exp}, \overline{exp}') \qquad \square$$

In the rest of this appendix, we will always consider cost expressions *exp* to be equivalent modulo the previous equalities that allow to move the *max* function around.

A.2. Upper bound correction

In this section, we prove that the cost we computed is a correct upper bound for the execution time of the program.

A.2.1. Properties of the runtime configuration's graph structure

The three following lemmas illustrate the fact that our definition of synchronization sets abstracts within synchronization sets all the complex artifacts of the point-to relation between objects, and exposes only a *simple* structure that we can analyze. Lemma A.3 states that the point-to relation between synchronization sets is a forest. Lemma A.4 shows that links in the point-to relation between objects of different synchronization sets have all different sources and targets. And Lemma A.5 shows that our notion of synchronization set is stable over time, in the sense that two different synchronization sets will never get merged during the runtime configuration's execution. This last lemma is very important as objects of two different synchronization sets are considered to execute in parallel in our analysis, while objects within the same synchronization set are considered to execute sequentially. Hence while splitting a synchronization set reduces the computed cost of the program (as objects considered to execute sequentially are now considered to execute in parallel), merge synchronization set increases it and could likely mean that our analysis did not compute the right cost before the merger.

Lemma A.3. *The synchronization graph of a runtime configuration cn is a forest.*

Proof. This is a direct consequence of merging in the same equivalence classes the nodes that break acyclicity in the point-to graph. $\square$

Lemma A.4. *Suppose given a runtime configuration cn,* $(x : f) \in cn$, $(x' : f') \in cn$ *and* $y \in cn$ *such that* $f \to y \in cn$, $f' \to y \in cn$ *and* $\mathcal{S}_{cn}(x) \neq \mathcal{S}_{cn}(y)$. *Then,* $x = x'$ *and* $f = f'$.

Proof. This is a direct consequence of the construction of the synchronization graph, where we quotient the point-to graph of cn by H, putting every actor accessible from different processes in the same equivalence class. $\square$

Lemma A.5. *Given a runtime configuration cn and a process* $p = s; f$ *such that* $cn = act(x, p', q)\ cn'$ *with* $p \in p'\ q$. *Then, for all* $A_1 \neq A_2 \in \mathcal{S}_{cn}$, *there exists* $A'_1 \neq A'_2 \in \texttt{sschem}(\mathcal{S}_{cn}, s)$ *with* $A_1 \subset A'_1$ *and* $A_2 \subset A'_2$.

Proof. This is directly proven by induction on s, and by the fact that $\mathcal{S}_{cn}$ is constructed by taking into account the dependency between actors in p. $\square$

Finally, the following definition and lemma show that the synchronization dependencies between processes have a simple tree structure. This property is important as when a process f synchronizes with a process f', our analysis counts the cost of f' within f. Hence, if the dependencies between process was a more general graph and several processes could synchronize with f', then its cost would be counted several times.

Definition A.9. The *process graph* of a runtime configuration cn is a graph (V, E) such that $V = \{f \mid f \in cn(x), x \in cn\}$ and $E = \{(f, f') \mid f, f' \in V \wedge f \to f' \in cn\}$.

Lemma A.6. *The process graph G of a runtime configuration cn is a tree.*

Proof. By induction on the reduction steps to reach cn from an initial configuration. Note that only the Async-Call reduction rule creates a link between futures; moreover, it creates this link to a fresh future name, so it is not possible to create loop. Hence, the process graph is a forest. Finally, because all futures must be synchronized, it is not possible to break a link towards a running process. Therefore, all nodes in the process graph are reachable. Thus, the process graph is a tree. $\square$

A.2.2. Properties of runtime evolution

The following Lemma simply states that the only part of I and Ψ relevant for the computation of the cost of a statement s is the one about the future on which s will synchronize. This lemma is useful, as it states that if two processes f and f' have no dependencies between each other, then executing f will not change the cost of f'.

Lemma A.7. *Suppose given $\mathcal{S}$, I_1, I_2, Ψ, x, l, t, s such that: the sets of parameters $\mathcal{S}$, I_i, Ψ, x, l, t and s are consistent, for $1 \leq i \leq 2$; and for all f free in s, we have $f \in \mathrm{dom}(I_1) \cap \mathrm{dom}(I_2)$ and $I_1(f) = I_2(f)$. Then, we have*

$$T_{\mathcal{S}}(I_1, \Psi, x, l, t, s) = T_{\mathcal{S}}(I_2, \Psi, x, l, t, s)$$

Proof. This is directly proven by induction on s, by remarking that all futures that are accessed in I_i (with $1 \leq i \leq 2$) are the ones free in s (rules (4) and (5) of Fig. 3). □

The following lemma states two of the core properties of our analysis, and is used in all the rest of this appendix. First, this lemma discusses the structure of the cost expression generated by our analysis. Looking at the rules (3), (4), (5) of Fig. 3, and using Lemma A.2, we can see that the cost of every synchronization set in Ψ has two parts: one that captures all the costs related to the execution of the analyzed statement (or *local cost*); and one that only refers to the cost of synchronization set's processes, plus the initial time ε of rule (3) (or *external cost*). More precisely, for every synchronization sets $A \in \mathrm{dom}(\Psi)$, we have $[\![\Psi(A)]\!] = max(exp^{A_1}, exp^{A_2})$ where exp^{A_1} is the first part of the cost, and exp^{A_2} is its second part. Consequently, with rules (4) and (5) of Fig. 3, we can remark that the computed cost equation has the same structure.

Second, this lemma illustrates how changing the starting moment of the analyzed statement is integrated in its cost. Consider a moment t_1 preceding by k units of time the moment t_2: as time advance is triggered by the program waiting, these two moments can be described as costs, following the structure we just discussed. Hence, t_1 can be written as $max(exp_1, exp_2)$ where exp_1 is the local cost and exp_2 the external cost, while t_2 can be written as $max(exp_1 + k, exp_2)$ (time passes locally). The cost expression in the translation environment follows the same pattern. Starting from these initial times, the lemma shows that if the statement s start at time t_2 instead of t_1, its computed cost will have the same shape, but its local cost will have k additional units of time.

Lemma A.8. *Suppose given $\mathcal{S}$, I, Ψ_1, Ψ_2, x, exp_1, exp_2, l, s and k with: $\mathcal{S}$, I, Ψ_i, x, t_i and l are coherent parameters for $1 \leq i \leq 2$ where $t_1 = max(exp_1, exp_2)$ and $t_2 = max(exp_1 + k, exp_2)$; $\mathrm{dom}(\Psi_1) = \mathrm{dom}(\Psi_2)$; and for all $A \in \mathrm{dom}(\Psi_1)$, there exists exp^{A_1} and exp^{A_2} such that $[\![\Psi_2(A)]\!] = max(exp^{A_1} + k, exp^{A_2})$ where $[\![\Psi_1(A)]\!] = max(exp^{A_1}, exp^{A_2})$. We then have*

$$T_{\mathcal{S}}(I, \Psi_2, x, l, t_2, s) = max(k + exp^{r_1}, exp^{r_2})$$
$$\text{with} \begin{cases} T_{\mathcal{S}}(I, \Psi_1, x, l, t_1, s) = max(exp^{r_1}, exp^{r_2}) \\ exp^{r_1} = max(exp_1 + exp_1{}', max_{A \in \mathrm{dom}(\Psi)}(exp^{A_1} + exp^{A_1}{}')) \text{ for some } exp_1{}' \text{ and } exp^{A_1}{}' \\ exp^{r_2} = max(exp_2 + exp_2{}', max_{A \in \mathrm{dom}(\Psi)}(exp^{A_2} + exp^{A_2}{}')) \text{ for some } exp_2{}' \text{ and } exp^{A_2}{}' \end{cases}$$

Proof. We prove this result by induction on s:

(i) Case $s = 0$. By construction, the lemma holds: I, Ψ_1 and Ψ_2 are empty in this case (all the futures must be synchronized at the end of a method), $e' = 0$ (for the same reason), and

$$\mathrm{T}_{\mathcal{S}}(\emptyset, \emptyset, x, 0, t_2, 0) = t_2$$
$$\mathrm{T}_{\mathcal{S}}(\emptyset, \emptyset, x, 0, t_1, 0) = t_1$$

(ii) Case $s = \nu x\,; s'$. By construction, we have that

$$\mathrm{T}_{\mathcal{S}}(I, \Psi_2, x, l, t_2, s) = \mathrm{T}_{\mathcal{S}}(I, \Psi_2, x, l, t_2, s')$$
$$\mathrm{T}_{\mathcal{S}}(I, \Psi_1, x, l, t_1, s) = \mathrm{T}_{\mathcal{S}}(I, \Psi_1, x, l, t_1, s')$$

Hence, with the induction hypothesis, we conclude the case.

(iii) Case $s = \nu f\colon \mathtt{m}(y, \overline{z}, \overline{e})\,; s'$ with $y \in \mathcal{S}(x)$. By rule (2) of Fig. 3, we have that

$$\mathrm{T}_{\mathcal{S}}(I, \Psi_2, x, l, t_2, s) = \mathrm{T}_{\mathcal{S}}(I[f \mapsto \mathcal{S}(x)], \Psi_2, x, l + \mathtt{m}(\overline{e}), t_2, s')$$
$$\mathrm{T}_{\mathcal{S}}(I, \Psi_1, x, l, t_1, s) = \mathrm{T}_{\mathcal{S}}(I[f \mapsto \mathcal{S}(x)], \Psi_1, x, l + \mathtt{m}(\overline{e}), t_1, s')$$

Hence, with the induction hypothesis, we conclude the case.

(iv) Case $s = \nu f\colon \mathtt{m}(y, \overline{z}, \overline{e})\,;s'$ with $y \notin \mathcal{S}(x)$. Let $S = \mathcal{S}(y)$. By rule (3) of Fig. 3, we have that

$$\begin{array}{l}
\mathtt{T}_{\mathcal{S}}(I, \Psi_2, x, l, t_2, s) = \mathtt{T}_{\mathcal{S}}(I[f \mapsto S], \Psi_2^f, x, l, t_2, s')\\
\mathtt{T}_{\mathcal{S}}(I, \Psi_1, x, l, t_1, s) = \mathtt{T}_{\mathcal{S}}(I[f \mapsto S], \Psi_1^f, x, l, t_1, s')\\
\quad \text{with } \Psi_2^f = \Psi_2[S \mapsto \varepsilon_2 \cdot \langle \mathtt{m}(\overline{e}), 0\rangle],\ \Psi_1^f = \Psi_1[S \mapsto \varepsilon_1 \cdot \langle \mathtt{m}(\overline{e}), 0\rangle]\\
\quad \text{and } \begin{cases} \varepsilon_2 = \Psi_2(S),\ \varepsilon_1 = \Psi_1(S) & \text{if } S \in \mathrm{dom}(\Psi_1)\\ \varepsilon_2 = t_2,\ \varepsilon_1 = t_1 & \text{else} \end{cases}
\end{array}$$

It is easy to see that $\mathrm{dom}(\Psi_2^f) = \mathrm{dom}(\Psi_1^f)$ and that

$$\begin{cases} [\![\Psi_2^f(S)]\!] = max(k + exp^{S_1}, exp^{S_2}), [\![\Psi_1^f(S)]\!] = max(exp^{S_1}, exp^{S_2}) & \text{if } S \in \mathrm{dom}(\Psi_1)\\ [\![\Psi_2^f(S)]\!] = max(k + exp_1, exp_2), [\![\Psi_1^f(S)]\!] = max(exp_1, exp_2) & \text{else} \end{cases}$$

Hence, with the induction hypothesis, we conclude the case.

(v) Case $s = f^{\checkmark}\,;s'$ with $x \in I(f)$. By rule (4) of Fig. 3, we have that

$$\begin{array}{l}
\mathtt{T}_{\mathcal{S}}(I, \Psi_2, x, l, t_2, s) = \mathtt{T}_{\mathcal{S}}(I, (\Psi_2 + l), x, 0, t_2 + l, s')\\
\mathtt{T}_{\mathcal{S}}(I, \Psi_1, x, l, t_1, s) = \mathtt{T}_{\mathcal{S}}(I, (\Psi_1 + l), x, 0, t_1 + l, s')
\end{array}$$

By construction, it is straightforward to see that $\mathrm{dom}(\Psi_2 + l) = \mathrm{dom}(\Psi_1 + l)$ and that for all $A \in \mathrm{dom}(\Psi_1 + l)$, we have

$$\begin{cases} [\![(\Psi_1 + l)(A)]\!] = max(exp^{A_1} + l, exp^{A_2} + l)\\ [\![(\Psi_2 + l)(A)]\!] = max(k + exp^{A_1} + l, exp^{A_2} + l) \end{cases}$$

Hence, with the induction hypothesis we conclude the case.

(vi) Case $s = f^{\checkmark}\,;s'$ with $x \notin I(f)$. Let $S = I(f)$. By rule (5) of Fig. 3, we have that

$$\begin{array}{l}
\mathtt{T}_{\mathcal{S}}(I, \Psi_2, x, l, t_2, s) = \mathtt{T}_{\mathcal{S}}(I \setminus F, \Psi_2^f, x, 0, exp^3, s')\\
\mathtt{T}_{\mathcal{S}}(I, \Psi_1, x, l, t_1, s) = \mathtt{T}_{\mathcal{S}}(I \setminus F, \Psi_1^f, x, 0, exp^4, s')\\
\quad \text{with } F = \{f' \mid I(f') = \mathcal{S}(x) \text{ or } I(f') = I(f)\}\\
\quad \text{and } \begin{cases} exp^3 = max(t_2 + l, [\![\Psi_2(I(f))]\!]),\ exp^4 = max(t_1 + l, [\![\Psi_1(I(f))]\!])\\ \Psi_2^f = (\Psi_2 \parallel exp^3) \setminus I(f),\ \Psi_1^f = (\Psi_1 \parallel exp^4) \setminus I(f) \end{cases}
\end{array}$$

It is straightforward to see that the following equalities hold:

$$\begin{cases} exp^3 = max(k + max(exp_1 + l, exp^{I(f)_1}), max(exp_2 + l, exp^{I(f)_2}))\\ exp^4 = max(max(exp_1 + l, exp^{I(f)_1}), max(exp_2 + l, exp^{I(f)_2})) \end{cases}$$

Additionally, we have that $\mathrm{dom}(\Psi_2^f) = \mathrm{dom}(\Psi_1^f)$, and let us consider $A \in \mathrm{dom}(\Psi_1^f)$, we then have

$$\begin{cases} [\![\Psi_2^f(A)]\!] = max(k + max(exp^{A_1}, exp_1 + l, exp^{I(f)_1}), max(exp^{A_2}, exp_2 + l, exp^{I(f)_2}))\\ [\![\Psi_1^f(A)]\!] = max(max(exp^{A_1}, exp_1 + l, exp^{I(f)_1}), max(exp^{A_2}, exp_2 + l, exp^{I(f)_2})) \end{cases}$$

Hence, with the induction hypothesis, we conclude the case.

(vii) Case $s = wait(e)\,;s'$. By rule (1) of Fig. 3, we have that

$$\begin{array}{l}
\mathtt{T}_{\mathcal{S}}(I, \Psi_2, x, l, t_2, s) = \mathtt{T}_{\mathcal{S}}(I, \Psi_1 + e, x, l, t_2 + e, s')\\
\mathtt{T}_{\mathcal{S}}(I, \Psi_1, x, l, t_1, s) = \mathtt{T}_{\mathcal{S}}(I, \Psi_1 + e, x, l, t_1 + e, s')
\end{array}$$

We can first remark that $t_2 + e = max(k + exp_1 + e, exp_2 + e)$ and $t_1 + e = max(exp_1 + e, exp_2 + e)$. By construction, it is straightforward to see that $\mathrm{dom}(\Psi_2 + e) = \mathrm{dom}(\Psi_1 + e)$. We then have

$$\begin{cases} [\![(\Psi_1 + e)(A)]\!] = max(exp^{A_1} + e, exp^{A_2} + e)\\ [\![(\Psi_2 + e)(A)]\!] = max(k + exp^{A_1} + e, exp^{A_2} + e) \end{cases}$$

Hence, with the induction hypothesis, we conclude the case. □

Lemma A.9. *Suppose given $\mathcal{S}$, I, Ψ, x, e, s, t and t' with: $\mathcal{S}$, I, Ψ, x, e, 0 and s are coherent parameters; for all $A \in \mathrm{dom}(\Psi)$, we have $\Psi(A) = 0 \cdot \langle exp_x^A, 0\rangle$; and $0 < t' \leq t$. We then have*

$$\begin{array}{l}
T_{\mathcal{S}}(I, \Psi, x, exp_x, 0, wait(t)\,;s) = max(t' + exp_x^1, max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + exp_x^A))\\
\quad with\ max(exp_x^1, max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + exp_x^A)) = T_{\mathcal{S}}(I, \Psi, x, exp_x, 0, wait(t - t')\,;s)
\end{array}$$

Proof. By rule (7) of Fig. 3, we have that

$$\mathtt{T}_{\mathcal{S}}(I,\Psi,x,exp_x,0,wait(t)\,;s)=\mathtt{T}_{\mathcal{S}}(I,\Psi+t,x,exp'_x,t,s)$$
$$\mathtt{T}_{\mathcal{S}}(I,\Psi,x,exp_x,0,wait(t-t')\,;s)=\mathtt{T}_{\mathcal{S}}(I,\Psi+(t-t'),x,exp'_x,(t-t'),s)$$

Hence, this lemma is a direct consequence of Lemma A.8. □

Lemma A.10. *Suppose given $\mathcal{S}$, I, Ψ, x, e, f and s with: $\mathcal{S}$, I, Ψ, x, 0 e, and s are coherent parameters; for all $A\in\mathrm{dom}(\Psi)$, we have $\Psi(A)=0\cdot\langle e^A,0\rangle$; and $I(f)=\mathcal{S}(x)$. We then have*

$$T_{\mathcal{S}}(I,\Psi,x,e,0,f^{\checkmark}\,;s)=max(e+e',max_{A\in\mathrm{dom}(\Psi)}(\Psi(A)+e^A))$$
$$\text{with } max(e',max_{A\in\mathrm{dom}(\Psi)}(\Psi(A)+e^A))=T_{\mathcal{S}}(I,\Psi,x,e,0,s)$$

Proof. By rules (4) and (7) of Fig. 3, we have that

$$\mathtt{T}_{\mathcal{S}}(I,\Psi,x,e,0,f^{\checkmark}\,;s)=\mathtt{T}_{\mathcal{S}}(I,\Psi+e,x,0,e,s)$$

This result is thus a direct consequence of Lemma A.8. □

Lemma A.11. *Suppose given $\mathcal{S}$, I, Ψ, x, e, f and s with: $\mathcal{S}$, I, Ψ, x, e, 0 and s are coherent parameters; for all $A\in\mathrm{dom}(\Psi)$, we have $\Psi(A)=0\cdot\langle e^A,0\rangle$; and $I(f)\neq\mathcal{S}(x)$. We then have*

$$T_{\mathcal{S}}(I,\Psi,x,e,0,f^{\checkmark}\,;s)=max(max(e,[\![\Psi(I(f))]\!])+e',max_{A\in\mathrm{dom}(\Psi)}(\Psi(A)+e^A))$$
$$\text{with }\begin{cases} max(e',max_{A\in\mathrm{dom}(\Psi)}(\Psi(A)+e^A))=T_{\mathcal{S}}(I\setminus F,\Psi\setminus I(f),x,0,0,s)\\ F=\{f'\in\mathrm{dom}(I)\mid I(f')=\mathcal{S}(x)\ or\ I(f')=I(f)\}\end{cases}$$

Proof. Let $S=I(f)$. By rule (5) of Fig. 3, we have that

$$\mathtt{T}_{\mathcal{S}}(I,\Psi,x,e,0,f^{\checkmark}\,;s)=\mathtt{T}_{\mathcal{S}}(I\setminus F,\Psi',x,0,e',s)$$
$$\text{with } F=\{f'\in\mathrm{dom}(I)\mid I(f')=\mathcal{S}(x)\text{ or }I(f')=S\}$$
$$\text{and } e'=max(e,[\![\Psi(S)]\!]),\ \Psi'=(\Psi\parallel e')\setminus S$$

The result can thus be proven in a similar way to Lemma A.8. □

Lemma A.12. *Suppose given $\mathcal{S}$, I, Ψ, x, exp_x, f and s with: $\mathcal{S}$, I, Ψ, x, e, 0 and s are coherent parameters; for all $A\in\mathrm{dom}(\Psi)$, we have $\Psi(A)=0\cdot\langle e^A,0\rangle$; and $f\in\mathrm{dom}(I)$. We then have*

$$T_{\mathcal{S}}(I,\Psi,x,e,0,f^{\checkmark}\,;s)\geq T_{\mathcal{S}}(I\setminus\{f\},\Psi,x,e,0,f^{\checkmark}\,;s)$$

Proof. This is directly proven by induction on s, by remarking that $\Psi(I(f))$ is a simple expression. □

A.2.3. Main results

Lemma A.13 *(Upper Bound Stability 1/3). Suppose given a cost solution Σ of the cost program generated from an* `alt` *program $P=\left(m_1(\overline{x}_1)=s_1,\cdots,m_n(\overline{x}_n)=s_n,s_{main}\right)$. Then Σ is also a cost solution of the runtime configuration $cn\stackrel{def}{=}act(start,s_{main}\,;f_{start},\emptyset)$.*

Proof. Let us consider the cost program eq_1 of the program P, and the cost configuration eq_2 of the runtime configuration cn. By construction, eq_1 and eq_2 are identical, except maybe for the entry f_{start}: we indeed have:

$$eq_1(f_{start})=\mathtt{translate}_{\mathtt{sschem}(\{\,\{start\}\,\},s_{main})}(\emptyset,\emptyset,start,0,0,s_{main})$$
$$eq_2(f_{start})=\mathtt{translate}_{\mathtt{sschem}(\mathcal{S}_{cn},s_{main})}(I_{cn},\Psi_{cn}\setminus\mathcal{S}_{cn}(start),start,0,0,s_{main})$$
$$\text{with }\begin{cases}\mathcal{S}_{cn}=\{\,\{start\}\,\}\\ I_{cn}=[f_{start}\mapsto\{start\}]\\ \Psi_{cn}=[\{start\}\mapsto f_{start}]\end{cases}$$

Remark that $\Psi_{cn}\setminus\mathcal{S}_{cn}(start)=\emptyset$, by Lemma A.7, we thus have that

$$\mathtt{translate}_{\mathtt{sschem}(\mathcal{S}_{cn},s_{main})}(I_{cn},\Psi_{cn}\setminus\mathcal{S}_{cn}(start),start,0,0,s_{main})$$
$$=\mathtt{translate}_{\mathtt{sschem}(\{\,\{start\}\,\},s_{main})}(\emptyset,\emptyset,start,0,0,s_{main})$$

which proves the result. □

Lemma A.14 *(Upper Bound Stability 2/3). Suppose given a runtime configuration cn such that there exists cn′ with* $cn \xrightarrow{t} cn'$. *Suppose moreover given a solution* Σ *of the cost configuration of cn. Then, there exists a solution* Σ' *of the cost configuration of cn′ such that for all* $f \in \mathrm{dom}(\Sigma)$, *we have* $\Sigma'(f) + t \leq \Sigma(f)$.

Proof. Let eq_1 be the cost configuration of cn and eq_2 the cost configuration of cn': we remark that $\mathrm{dom}(eq_1) = \mathrm{dom}(eq_2)$, and we first prove by induction on cn that for all $f \in \mathrm{dom}(eq_1)$, we either have that:

$$eq_1(f) = max(t' + e^1, max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + e^A))$$
$$\text{with } eq_2(f) = max(e^1, max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + e^A))$$

or $eq_1(f) = eq_2(f) = \sum_{f \in F} f + e$ for some e and with $F = \{f' \mid f \to f' \in cn\} \neq \emptyset$. If the configuration is empty, then its cost configuration is empty and the result trivially holds. Now let us consider that cn is not empty. By the definition of $cn \xrightarrow{t} cn$, triggered by the rule (Tick), cn is strongly t-*stable* (see Definition 2.1). Let us consider $f \in \mathrm{dom}(\Sigma)$, we have three cases:

- Case $cn = act(x, wait(e); s; f, q)\ cn''$. By Definition 2.1, we have $cn' = act(x, wait(k); s; f, q)\ \Phi(cn'', t)$ with $k = [\![e]\!] - t$. By Lemma A.9, we have that $eq_1(f) = max(t' + e^1, max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + e^A))$ with $eq_2(f) = max(e^1, max_{A \in \mathrm{dom}(\Psi)}(\Psi(A) + e^A))$.
- Case $cn = act(x, f'^{\checkmark}; s; f, q)\ cn'$ with $I_{cn}(f) = I_{cn}(f')$. By Definition 2.1, we have $cn' = act(x, f'^{\checkmark}; s; f, q)\ \Phi(cn'', t)$, which implies that $eq_1(f) = eq_2(f)$. Using Lemma A.8, it is straightforward to see that $eq_2(f) = \sum_{f \in F} f + e$ for some e, with $F = \{f'' \mid I_{cn}(f) = I_{cn}(f''),\ f \to f'' \in cn\}$ and $f' \in F$.
- Case $cn = act(x, f'^{\checkmark}; s; f, q)\ cn'$ with $I_{cn}(f) \neq I_{cn}(f')$. By Definition 2.1, we have $cn' = act(x, f'^{\checkmark}; s; f, q)\ \Phi(cn'', t)$, which implies that $eq_1(f) = eq_2(f)$. Using Lemma A.8, it is straightforward to see that $eq_2(f) = \sum_{f \in F} f + e$ for some e, with $F = \{f'' \mid I_{cn}(f') = I_{cn}(f''),\ f \to f'' \in cn\}$ and $f' \in F$.

Using Lemma A.6, we have that the process graph G of cn is a tree, and so we can construct the solution Σ' inductively, starting from the leaves of G. □

Lemma A.15 *(Upper Bound Stability 3/3). Suppose given a configuration cn such that there exists cn′ with* $cn \to cn'$. *Suppose moreover given a solution* Σ *of the cost configuration of cn. Then, there exists a solution* Σ' *of the cost configuration of cn′ such that for all* $f \in \mathrm{dom}(\Sigma)$, *we have* $\Sigma'(f) \leq \Sigma(f)$.

Proof. Let us consider the cost configuration eq_1 of cn and eq_2 of cn'. We prove that for all $f \in \mathrm{dom}(eq_1)$, we have $eq_2(f) \leq eq_1(f)$, which thus proves the result. We construct our demonstration by case distinction on the reduction rule, modulo the rule Context.

- Case New. It is straightforward to see that, with the definition of $\mathcal{S}_{cn}$, that $eq_1 = eq_2$. Hence, the result holds for the case.
- Case Get-True. We have that $cn = act(x, f_1^{\checkmark}; p, q)\ cn'$, with $p = s; f$. We can first remark that for all $(x : f') \in cn$ such that $f' \neq f$, we have that $eq_1(f) = eq_2(f)$. Moreover, from the construction of the future localization, we have that $f_1 \notin \mathrm{dom}(I_{cn})$. Hence, using the rule (6) of Fig. 3, we have that $eq_1(f) = eq_2(f)$.
- Case Get-False. It is easy in this case to see that $eq_1 = eq_2$, which gives us the result.
- Case Async-Call. We have that $cn = act(x, \nu f_1{:}\ \mathtt{m}(z, \overline{e}); p, q)\ cn'$, with $p = s; f$. Let f_2 be the fresh future name created in this rule. We now have two sub-cases.
  Let first consider that $\mathcal{S}_{cn}(z) = \mathcal{S}_{cn}(x)$. By construction, we have $\Psi_{cn'}(\mathcal{S}_{cn'}(z)) = \Psi_{cn}(\mathcal{S}_{cn}(z))$, as f_2 is free in p. Moreover, considering the rule (7) of Fig. 8, for all $f' \in \mathrm{dom}(eq_1) \setminus \{f_2\}$, we have $eq_1(f') = eq_2(f')$. Now, by rule (4) of Fig. 8, we have that $eq_2(f_2) = \mathtt{m}(z, \overline{e})$, and $eq_1(f) = eq_2(f)\{^{\mathtt{m}(z,\overline{e})}/_{f_2}\}$. By stating that $\Sigma' = \Sigma[f_2 \mapsto \Sigma(\mathtt{m}(z, \overline{e}))]$, we have that $\Sigma(eq_1(f)) = \Sigma'(eq_2(f))$, which concludes this sub-case.
  Second, consider that $\mathcal{S}_{cn}(z) \neq \mathcal{S}_{cn}(x)$. By Lemma A.4, we have that f is the only future name such that $f \to z \in cn$. Hence, by construction, we have that for all $f' \in (\mathrm{dom}(eq_1)) \setminus \{f\}$, we have $eq_1(f') = eq_2(f')$. Now, by rule (4) of Fig. 8, we have that $eq_2(f_2) = \mathtt{m}(z, \overline{e})$, and $eq_1(f) = eq_2(f)\{^{\mathtt{m}(z,\overline{e})}/_{f_2}\}$. By stating that $\Sigma' = \Sigma[f_2 \mapsto \Sigma(\mathtt{m}(z, \overline{e}))]$, we have that $\Sigma(eq_1(f)) = \Sigma'(eq_2(f))$, which concludes this sub-case.
- Case Bind-Fun. Here, we have that $eq_1(f) = \mathtt{m}(x, \overline{v})$, and $eq_2(f) = e$ where $eq_1(\mathtt{m}(x, \overline{v})) = e$. Thus, we conclude the case.
- Case Wait-0. It is easy in this case to see that $eq_1 = eq_2$, which gives us the result.
- Case Activate. It is easy in this case to see that $eq_1 = eq_2$, which gives us the result.
- Case Return. We have that $cn = act(x, f, q)\ cn'$. By construction, we have that $eq_1(f) = 0 = eq_2(f)$ (by rule (2.2) of Fig. 8). By Lemma A.6, either: i) $f = f_{start}$, and so we have the result, as $eq_2(f) = 0$ for all $f \in \mathrm{dom}(eq_2)$; or ii) there exists exactly one f' such that $f' \to f \in cn$. By construction, for all $f'' \in \mathrm{dom}(I_{cn}) \setminus f'$, we have $eq_1(f'') = eq_2(f'')$. Finally, by Lemma A.12, we have that $eq_1(f') \geq eq_2(f')$, which concludes the case. □

Theorem A.16 *(Correction). Given an* `alt` *program P, and a solution* Σ *of the cost equation of P, we have that* $\Sigma(f_{start})$ *is an upper bound of the execution time of P.*

Proof. By induction on the reduction time of P, using the three previous lemma. □

References

[1] E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, G. Román-Díez, SACO: static analyzer for concurrent objects, in: Proceedings of TACAS 2014, in: Lect. Notes Comput. Sci., vol. 8413, Springer, 2014, pp. 562–567.
[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, Closed-form upper bounds in static cost analysis, J. Autom. Reason. 46 (2) (2011) 161–203.
[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of object-oriented bytecode programs, Theor. Comput. Sci. 413 (1) (2012) 142–159.
[4] E. Albert, J. Correas, E.B. Johnsen, K.I. Pun, G. Román-Díez, Parallel cost analysis, ACM Trans. Comput. Log. 19 (4) (Nov. 2018) 31:1–31:37.
[5] S. Blazy, A. Maroneze, D. Pichardie, Formal verification of loop bound estimation for WCET analysis, in: Proceedings of VSTTE'13, in: Lect. Notes Comput. Sci., vol. 8164, Springer, 2013, pp. 281–303.
[6] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, J. Giesl, Alternating runtime and size complexity analysis of integer programs, in: Proceedings of TACAS, in: Lect. Notes Comput. Sci., vol. 8413, Springer, 2014, pp. 140–155.
[7] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility, Future Gener. Comput. Syst. 25 (6) (2009) 599–616.
[8] A. Flores Montoya, R. Hähnle, Resource analysis of complex programs with cost equations, in: Proceedings of APLAS 2014, in: Lect. Notes Comput. Sci., vol. 8858, Springer, 2014, pp. 275–295.
[9] E. Giachino, E.B. Johnsen, C. Laneve, K. I Pun, Time complexity of concurrent programs, in: Proceedings of FACS 2015, Springer, 2016, pp. 199–216.
[10] S. Gulwani, K.K. Mehra, T. Chilimbi, Speed: precise and efficient static estimation of program computational complexity, in: ACM SIGPLAN Not., vol. 44, ACM, 2009, pp. 127–139.
[11] J. Hoffmann, K. Aehlig, M. Hofmann, Multivariate amortized resource analysis, ACM Trans. Program. Lang. Syst. 34 (3) (2012) 14.
[12] J. Hoffmann, Z. Shao, Automatic static cost analysis for parallel programs, in: J. Vitek (Ed.), Programming Languages and Systems, Springer, Berlin Heidelberg, 2015, pp. 132–157.
[13] E.B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: a core language for abstract behavioral specification, in: Proceedings of FMCO 2010, in: Lect. Notes Comput. Sci., vol. 6957, Springer, 2011, pp. 142–164.
[14] C. Laneve, G. Zavattaro, Foundations of web transactions, in: Proceedings of FOSSACS, in: Lect. Notes Comput. Sci., vol. 3441, Springer, 2005, pp. 282–298.
[15] G. Morrisett, D. Walker, K. Crary, N. Glew, From system f to typed assembly language, ACM Trans. Program. Lang. Syst. 21 (3) (1999) 527–568.
[16] P.W. Trinder, M.I. Cole, K. Hammond, H. Loidl, G. Michaelson, Resource analyses for parallel and distributed coordination, Concurr. Comput. 25 (3) (2013) 309–348.
[17] B. Wegbreit, Mechanical program analysis, Commun. ACM 18 (9) (1975) 528–539.